www.sapientia.uji.es | 151

Informática Gráfica

2ª edición

José Ribelles
Ángeles López



Col·lecció «Sapientia», núm. 151

INFORMÁTICA GRÁFICA

2ª EDICIÓN

José Ribelles
Ángeles López

Departamento de Lenguajes y Sistemas Informáticos

■ Codi d'assignatura: Informática Gráfica (VJ1221)



Edita: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions
Campus del Riu Sec. Edifici Rectorat i Serveis Centrals. 12071 Castelló de la Plana
http://www.tenda.uji.es e-mail: publicacions@uji.es

Colección Sapientia 151
www.sapientia.uji.es

Primera edición, 2015
Segunda edición, 2019

ISBN: 978-84-17429-87-4
DOI: http://dx.doi.org/10.6035/Sapientia151



Publicacions de la Universitat Jaume I es miembro de la UNE, lo que garantiza la difusión y comercialización de sus publicaciones a nivel nacional e internacional. www.une.es.



Este libro, de contenido científico, ha estado evaluado por personas expertas externas a la Universitat Jaume I, mediante el método denominado revisión por iguales, doble ciego.

Índice general

Prefacio **19**

1. Introducción a WebGL **21**
1.1. Antecedentes 22
1.2. Prueba de WebGL 23
1.3. El mínimo programa 24
1.4. El primer triángulo 28
1.4.1. El *pipeline* 31
1.4.2. GLSL 31

2. Modelado poligonal **35**
2.1. Representación 36
2.1.1. Caras independientes 37
2.1.2. Vértices compartidos 37
2.1.3. Tiras y abanicos de triángulos 39
2.2. La normal 41
2.3. Mallas y WebGL 43
2.3.1. Preparación y dibujado 44
2.3.2. Tipos de primitivas geométricas 44
2.3.3. Variables *uniform* 50
2.3.4. Variables *out* 51

3. Transformaciones geométricas **55**
3.1. Transformaciones básicas 56
3.1.1. Traslación 56
3.1.2. Escalado 56
3.1.3. Rotación 57
3.2. Concatenación de transformaciones 57
3.3. Matriz de transformación de la normal 58
3.4. Giro alrededor de un eje arbitrario 61
3.5. La biblioteca GLMATRIX 61
3.6. Transformaciones en WebGL 63

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

4. Viendo en 3D **69**
4.1. Transformación de la cámara 69
4.2. Transformación de proyección 71
4.2.1. Proyección paralela 71
4.2.2. Proyección perspectiva 73
4.3. Transformación al área de dibujo 74
4.4. Eliminación de partes ocultas 74
4.5. Viendo en 3D con WebGL 75

5. Modelos de iluminación y sombreado **81**
5.1. Modelo de iluminación de Phong 82
5.1.1. Luz ambiente 83
5.1.2. Reflexión difusa 83
5.1.3. Reflexión especular 84
5.1.4. Atenuación 86
5.1.5. Materiales 87
5.1.6. El modelo de Phong 87
5.2. Tipos de fuentes de luz 89
5.3. Modelos de sombreado 91
5.4. Implementación de Phong con WebGL 93
5.4.1. Normales en los vértices 93
5.4.2. Materiales 95
5.4.3. Fuente de luz 96
5.5. Iluminación por ambas caras 98
5.6. Sombreado cómic 98
5.7. Niebla 100

6. Texturas **105**
6.1. Coordenadas de textura 105
6.1.1. Repetición de la textura 109
6.2. Leyendo téxeles 111
6.2.1. Magnificación 112
6.2.2. Minimización 112
6.3. Texturas en WebGL 113
6.4. Texturas 3D 116
6.5. Mapas de cubo 117
6.5.1. *Reflection mapping* 117
6.5.2. *Refraction mapping* 119
6.5.3. *Skybox* 119
6.6. *Normal mapping* 123
6.7. *Displacement mapping* 123
6.8. *Alpha mapping* 124

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

7. Texturas procedurales **127**
7.1. Rayado . . . 128
7.2. Damas . . . 130
7.3. Enrejado . . . 132
7.4. Ruido . . . 133

8. Realismo visual **139**
8.1. Transparencia . . . 139
8.2. Espejos . . . 143
8.2.1. Obtener la matriz de simetría . . . 144
8.2.2. Evitar dibujar fuera de los límites . . . 144
8.3. Sombras . . . 147
8.3.1. Sombras proyectivas . . . 147
8.3.2. *Shadow mapping* . . . 149

9. Interacción y animación con *shaders* **151**
9.1. Selección de objetos . . . 151
9.1.1. Utilizando el propio canvas . . . 152
9.1.2. Utilizando un FBO . . . 153
9.2. Animación . . . 154
9.2.1. Eventos de tiempo . . . 155
9.2.2. Encendido / apagado . . . 155
9.2.3. Texturas . . . 156
9.2.4. Desplazamiento . . . 157
9.3. Sistemas de partículas . . . 158

10. Procesamiento de imagen **163**
10.1. Apariencia visual . . . 163
10.1.1. *Antialiasing* . . . 163
10.1.2. Corrección gamma . . . 166
10.2. Posproceso de imagen . . . 167
10.2.1. Brillo . . . 167
10.2.2. Contraste . . . 168
10.2.3. Saturación . . . 168
10.2.4. Negativo . . . 168
10.2.5. Escala de grises . . . 170
10.2.6. Convolución . . . 170
10.3. Transformaciones . . . 171

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Índice de figuras

1.1. Ejemplo de escena tridimensional dibujada con WebGL 21
1.2. Ejemplos de objetos dibujados mediante *shaders* 22
1.3. Resultado con éxito del test de soporte de WebGL en un navegador (`http://get.webgl.org`) 23
1.4. Resultado de la página `http://webglreport.org` que proporciona información sobre algunas propiedades relativas al soporte de WebGL 2.0 en el navegador. 25
1.5. El primer triángulo con WebGL 30
1.6. Secuencia básica de operaciones del *pipeline* de WebGL 31

2.1. A la izquierda, objeto representado mediante cuadriláteros y a la derecha, objeto representado mediante triángulos 35
2.2. Representación poligonal de una copa y resultado de su visualización 36
2.3. Ejemplos de mallas poligonales, de izquierda a derecha, *Bunny*, *Armadillo* y *Dragon* (modelos cortesía del *Stanford Computer Graphics Laboratory*) 37
2.4. Esquema de almacenamiento de una malla poligonal mediante la estructura de caras independientes 38
2.5. Esquema de almacenamiento de una malla poligonal mediante la estructura de vértices compartidos 38
2.6. Ejemplo de abanico y tira de triángulos 40
2.7. Representación de la malla mediante una tira de triángulos generalizada 40
2.8. Visualización del modelo poligonal de una tetera. En la imagen de la izquierda se pueden obervar los polígonos utilizados para representarla 42
2.9. Estrella dibujada a partir de los vértices de un pentágono 46
2.10. Estrella dibujada con líneas (izquierda) y con triángulos (derecha) 46
2.11. Ejemplo de dos estrellas: una aporta el color interior y la otra el color del borde 51
2.12. Resultado de visualizar un triángulo con un valor de color diferente para cada vértice 52
2.13. Observa el triángulo y contesta, ¿qué color se ha asignado a cada vértice? 53

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

3.1. Ejemplos de objetos creados utilizando transformaciones geométricas 55
3.2. En la imagen de la izquierda se representa de perfil un polígono y su normal n. En la imagen de la derecha se muestra el mismo polígono tras aplicar una transformación de escalado no uniforme $S(2, 1)$. Si se aplica esta transformación a la normal n, se obtiene p como vector normal en lugar de m, que es la normal correcta . . 58
3.3. La nueva base formada por los vectores d, e y f se transforma para coincidir con los ejes de coordenadas 61
3.4. Escena modelada a partir de la primitiva geométrica cubo 65
3.5. Ejemplo de una grúa de obra modelada con cubos y transformaciones geométricas 66
3.6. Tanque modelado a partir de diferentes primitivas geométricas . . 66
3.7. Reloj modelado a partir de primitivas geométricas básicas 66
3.8. Otros ejemplos de juegos de bloques de madera coloreados 67

4.1. Parámetros para ubicar y orientar una cámara: p, posición de la cámara; UP, vector de inclinación; i, punto de interés 70
4.2. Transformación de la cámara. La cámara situada en el punto p en la imagen de la izquierda se transforma para quedar como se observa en la imagen de la derecha. Dicha transformación se aplica al objeto de tal manera que lo que se observa sea lo mismo en ambas situaciones 70
4.3. Vista de un cubo obtenida con: (a) proyección perspectiva y (b) proyección paralela 71
4.4. Esquema del volumen de la vista de una proyección paralela . . . 72
4.5. Volumen canónico de la vista, cubo de lado 2 centrado en el origen de coordenadas 72
4.6. Esquema del volumen de la vista de una proyección perspectiva . 73
4.7. Ejemplo de canvas con tres áreas de dibujo o *viewports*. La escena es la misma en las tres áreas, sin embargo, la transformación de la cámara es distinta para cada área. 74
4.8. Ejemplo de escena visualizada: (a) sin resolver el problema de la visibilidad y (b) con el problema de la visibilidad resuelto 75
4.9. En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado de la primitiva 77
4.10. Ejemplo de tres áreas de dibujo o *viewports* en los que se especifica para cada uno de ellos su origen y tamaño 78
4.11. Si la proporción del volumen de la vista y del área de dibujo (*viewport*) coinciden no se producirá ningún tipo de deformación en la escena visualizada 78
4.12. El vértice del cono en la imagen de la derecha queda fuera del volumen de la vista y se puede apreciar el resultado de la operación de recortado 78

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

4.13. En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado del fragmento donde se indica que el test de profundidad se realiza con posterioridad a la ejecución del *shader* de fragmentos y de manera opcional 79

5.1. Ejemplo de las componentes del modelo de iluminación de Phong: (a) Luz ambiente; (b) Reflexión difusa; (c) Reflexión especular . . 82
5.2. Ejemplo obtenido utilizando el modelo de iluminación de Phong . 82
5.3. Ejemplos de iluminación: (a) Solo luz ambiente; (b) Luz ambiente y reflexión difusa;
(c) Luz ambiente, reflexión difusa y reflexión especular 83
5.4. Vectores involucrados en el cálculo de la reflexión difusa 84
5.5. Vectores involucrados en el cálculo de la reflexión especular . . . 85
5.6. Ejemplos de iluminación con diferentes valores de α para el cálculo de la reflexión especular 85
5.7. Vectores involucrados en el cálculo de la reflexión especular con el vector intermedio 86
5.8. Ejemplo de escena sin atenuación (izquierda) y con atenuación de la luz (derecha) 86
5.9. Ejemplos de materiales, de izquierda a derecha y de arriba a abajo: Oro, Cobre, Plata, Obsidiana, Jade, Rubí, Bronce, Turquesa. . . . 88
5.10. Ejemplo de escena iluminada: a la izquierda, con una luz posicional y a la derecha, con la fuente convertida en foco 89
5.11. Parámetros característicos de un foco de luz 90
5.12. Resultados obtenidos utilizando los diferentes modelos de sombreado 91
5.13. Ejemplos obtenidos con los modelos de sombreado: Plano, Gouraud y Phong 97
5.14. Ejemplo de modelo en el que hay que aplicar iluminación en ambas caras para una correcta visualización 98
5.15. Resultado de la función *toonShading* con diferentes valores de la variable *levels* 99
5.16. Resultados obtenidos utilizando el *shader* de niebla 101

6.1. Resultados obtenidos al aplicar diferentes texturas 2D sobre el mismo objeto 3D 106
6.2. Ejemplo de aplicar una textura 2D diferente a la misma escena . . 106
6.3. Resultados obtenidos utilizando diferentes coordenadas de textura 106
6.4. En el espacio de la textura, el rango de coordenadas válido es $[0, 1]$. En el espacio del objeto, cada fragmento recibe las coordenadas de textura interpoladas 107
6.5. Ejemplos de aplicación de textura 2D. En estos casos el color definitivo de un fragmento se ha obtenido a partir de la textura y del modelo de iluminación de Phong 108
6.6. Resultado de combinar dos texturas 108

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

6.7. Ejemplos de repetición de textura 109
6.8. Resultados de extensión del borde de la textura 109
6.9. Repetición de la textura de manera simétrica (imagen de la izquierda) y repetición normal como la de la figura 6.7 (imagen de la derecha) 110
6.10. Filtro caja de WebGL, devuelve el valor del téxel más cercano y produce el efecto de pixelado 112
6.11. Filtro bilineal de WebGL, devuelve la interpolación lineal de cuatro téxeles y produce el efecto de borrosidad 112
6.12. Ejemplo de escena con *Mipmapping* (izquierda) y solo filtro bilineal (derecha). 113
6.13. Ejemplos obtenidos utilizando texturas en WebGL 116
6.14. Ejemplos de texturas de mapa de cubo (fuente: Emil Persson `http://www.humus.name`) 117
6.15. Vectores involucrados en *reflection mapping* 118
6.16. El mapa de cubo (que se muestra a la derecha en la figura 6.14) se ha utilizado para simular que el toro está reflejando su entorno . . 118
6.17. Ejemplo de *refraction mapping* 120
6.18. Ejemplos obtenidos utilizando *reflection mapping* en WebGL . . . 121
6.19. Ejemplos obtenidos utilizando *refraction* y *reflection mapping* en WebGL. El índice de refracción es, de izquierda a derecha, de 0,95 y 0,99 122
6.20. Objetos texturados con la técnica de *bump mapping*. La modificación de la normal produce que aparentemente la superficie tenga bultos 123
6.21. Ejemplo de desplazamiento de la geometría 124
6.22. Ejemplos de aplicación de la técnica *alpha mapping* 124
6.23. Ejemplos de aplicación de diferentes *alpha maps* sobre el mismo objeto 125

7.1. Ejemplo de objeto dibujado con una textura procedural. En este caso, el valor devuelto por la función de textura se utiliza para determinar si hay que eliminar un determinado fragmento 127
7.2. Ejemplo de combinación de textura 2D y textura procedural. A la izquierda el objeto es dibujado sin textura, en el centro se le ha aplicado una textura 2D, y a la derecha se ha combinado la textura 2D con una textura procedural 128
7.3. Ejemplos del *shader* de rayado 129
7.4. Ejemplos del *shader* de rayado utilizando la otra coordenada de textura 129
7.5. Ejemplos del *shader* de rayado utilizando la función seno 129
7.6. Ejemplos del *shader* de damas 131
7.7. Ejemplo del *shader* de damas utilizando un factor de escala distinto para cada dirección 131

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

7.8. Ejemplos del *shader* de enrejado 132
7.9. Ejemplos de enrejados circulares 133
7.10. Ejemplo de enrejado circular en el que se ha eliminado la superficie del círculo 133
7.11. Ejemplos obtenidos utilizando una función de ruido como textura procedural 134
7.12. Ejemplos obtenidos mediante el uso de la función de ruido de Perlin en el *shader* de fragmentos 134
7.13. Ejemplos obtenidos con la función de ruido de Perlin 135

8.1. Tres ejemplos de transparencia con, de izquierda a derecha, *alfa* = 0.3, 0.5 y 0.7 140
8.2. Dos resultados diferentes en los que únicamente se ha variado el orden en el dibujado de los objetos transparentes 141
8.3. Los planos transparentes de la izquierda se han pintado utilizando la función *gl.ONE* mientras que en los de la derecha se ha utilizado *gl.ONE_MINUS_SRC_ALPHA* 142
8.4. Ejemplo de objeto transparente con, de izquierda a derecha, *alfa*= 0.3, 0.5 y 0.7, pintado utilizando el código recogido en el listado 8.2. 143
8.5. Ejemplo de objeto reflejado en una superficie plana 144
8.6. Al dibujar la escena simétrica es posible observarla fuera de los límites del objeto reflejante (izquierda). El *buffer* de plantilla se puede utilizar para resolver el problema (derecha) 145
8.7. Escena en la que el suelo hace de objeto espejo 145
8.8. En este ejemplo, la misma escena se interpreta de manera distinta dependiendo de si se añade o no la sombra 147
8.9. Ejemplo de *stiching* producido por ser coplanares el suelo y el objeto sombra 148
8.10. El objeto sombra supera los límites de la superficie sobre la que recae 148
8.11. Ejemplo de sombras proyectivas transparentes 149
8.12. Ejemplo de *shadow mapping*. A la izquierda se observa el mapa de profundidad obtenido desde la fuente de luz; a la derecha se muestra la escena con sus sombras 150

9.1. Las dos escenas pintadas para la selección de objetos 152
9.2. Objetos animados con la técnica de desplazamiento 158
9.3. Animación de un mosaico implementado como sistema de partículas 159
9.4. Animación de banderas implementada como sistema de partículas 159
9.5. Ejemplo de sistema de partículas dibujado con tamaños de punto diferentes 161

10.1. Ejemplo de procesado de imagen. A la imagen de la izquierda se le ha aplicado un efecto de remolino, generando la imagen de la derecha 164

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

10.2. En la imagen de la izquierda se observa claramente el efecto escalera, que se hace más suave en la imagen de la derecha 164
10.3. Ejemplo de funcionamiento del *supersampling* 165
10.4. Comparación del resultado de aplicar MSAA (izquierda) y no aplicarlo (derecha). 165
10.5. Esquema de funcionamiento de la correción gamma 166
10.6. Ejemplos de corrección gamma: 1.0 (izquierda) y 2.2 (derecha) . 166
10.7. Ejemplos de posproceso de imagen 167
10.8. Ejemplos de modificación del brillo de la imagen con factores de escala 0.9, 1.2 y 1.5 168
10.9. Ejemplos de modificación del contraste de la imagen: 0.5, 0.75 y 1.0 169
10.10Ejemplos de modificación de la saturación de la imagen: 0.2, 0.5 y 0.8 169
10.11Ejemplo del resultado del negativo de la imagen 169
10.12Ejemplo del resultado de la imagen en escala de grises 170
10.13Ejemplo de resultado de la operación de convolución con el filtro de detección de bordes en la parte izquierda de la imagen 171
10.14*Warping* de una imagen: imagen original en la izquierda, malla modificada en la imagen del centro y resultado en la imagen de la derecha 172

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Índice de listados

1.1. Ejemplo de creación de un canvas con HTML5 26
1.2. Código JAVASCRIPT para obtener un contexto WebGL 2.0 27
1.3. Código que utiliza órdenes de WebGL 27
1.4. Compilación y enlazado de un *shader* 29
1.5. Ejemplo básico de *shader* de vértices y *shader* de fragmentos . . . 30
2.1. Estructura correspondiente a la representación de caras independientes 37
2.2. Estructura correspondiente a la representación de vértices compartidos 38
2.3. Modelo de un cubo definido con triángulos dispuesto para ser utilizado con WebGL 43
2.4. Código mínimo que se corresponde con la estructura básica de un programa que utiliza WebGL (disponible en *c02/dibuja.js*) 47
2.5. Código HTML que incluye un canvas y los dos *shaders* básicos (disponible en *c02/dibuja.html*) 49
2.6. Ejemplo de variable *uniform* en el *shader* de fragmentos 50
2.7. Obtiene la referencia y establece el valor de la variable *uniform* `myColor` 50
2.8. Ejemplo de modelo con dos atributos por vértice: posición y color 51
2.9. Localización y habilitación de los dos atributos: posición y color . 51
2.10. Ejemplo de *shader* con dos atributos por vértice: posición y color . 52
2.11. Dibujo de un modelo con dos atributos por vértice 53
3.1. Visualización en alambre de un modelo poligonal definido con triángulos independientes 63
3.2. Ejemplo de los pasos necesarios para dibujar un objeto transformado 64
3.3. *Shader* de vértices que opera cada vértice con la matriz de transformación del modelo 64
4.1. Algoritmo del *z-buffer* 75
4.2. *Shader* de vértices para transformar la posición de cada vértice . . 76
5.1. Función que implementa para una fuente de luz el modelo de iluminación de Phong sin incluir el factor de atenuación 88
5.2. *Shader* para calcular la iluminación con un foco de luz 90
5.3. *Shader* para realizar un sombreado de Gouraud 92
5.4. *Shader* para realizar un sombreado de Phong 92

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

5.5. Modelo de un cubo con la normal definida para cada vértice . . . 93
5.6. Obtención de referencias para el uso de las normales 94
5.7. Funciones para el cálculo y la inicialización de la matriz de la normal en el *shader* a partir de la matriz modelo-vista 94
5.8. Nueva función de dibujo que incluye dos atributos: posición y normal 95
5.9. Obtención de las referencias a las variables del *shader* que contendrán el material 95
5.10. La función *setShaderMaterial* recibe un material como parámetro e inicializa las variables del *shader* correspondientes. En la función *drawScene* se establece un valor de material antes de dibujar el objeto 95
5.11. Obtención de las referencias a las variables del *shader* que contendrán los valores de la fuente de luz 96
5.12. La función *setShaderLight* inicializa las variables del *shader* correspondientes 96
5.13. Iluminación en ambas caras, modificación en el *shader* de fragmentos 98
5.14. Iluminación en ambas caras, modificación en el *shader* de fragmentos 99
5.15. *Shader* de niebla 100
6.1. Cambios necesarios para que un *shader* utilice una textura 2D . . 107
6.2. Cambios en el *shader* de fragmentos para utilizar varias texturas 2D 108
6.3. Creación de una textura en WebGL 2.0 114
6.4. Asignación de un objeto textura a una unidad de textura 115
6.5. Asignación de unidad a un *sampler2D* 115
6.6. Habilitación del atributo de coordenada de textura 115
6.7. Especificación de tres atributos: posición, normal y coordenadas de textura 116
6.8. Cambios en el *shader* para *reflection mapping* 119
6.9. Cambios en el *shader* para *refraction mapping* 120
6.10. Cambios en el *shader* para *skybox* 121
6.11. *Shader* para *skybox* y *reflection mapping* 122
7.1. *Shader* de rayado 130
7.2. *Shader* de damas 131
7.3. *Shader* de enrejado 132
8.1. Secuencia de operaciones para dibujar objetos transparentes . . . 141
8.2. Objetos transparentes 142
8.3. Secuencia de operaciones para dibujar objetos reflejantes 146
9.1. Conversión de coordenadas para ser utilizadas en WebGL 152
9.2. Acceso al color de un píxel en el *framebuffer* 153
9.4. Acceso al color de un píxel en el FBO 154
9.5. *Shader* para encendido / apagado 155
9.6. Función que controla el encendido / apagado 156
9.7. Función que actualiza el desplazamiento de la textura con el tiempo 157
9.8. *Shader* para actualizar las coordenadas de textura con el tiempo . 157
9.9. Cortina de partículas 160
9.10. Dibujado del sistema de partículas 160

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

9.11. *Shader* de vértices para el sistema de partículas 161
10.1. *Shader* de fragmentos para la correción gamma 167
10.2. Modificación del brillo de una imagen 168
10.3. Modificación del contraste de una imagen 168
10.4. Modificación de la saturación de la imagen 168
10.5. Negativo del fragmento 169
10.6. Cálculo de la imagen en escala de grises 170

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Prefacio

El objetivo de este libro es proporcionar un material teórico y práctico para apoyar la docencia, tanto presencial, desarrollada en clases de teoría o en laboratorio, como no presencial, proporcionando al estudiante un material que facilite el estudio de la materia, de un nivel y contenido adecuado para la asignatura Informática Gráfica del grado en Diseño y Desarrollo de Videojuegos de la Universitat Jaume I. Este libro pretende ser el complemento ideal a las explicaciones que el profesor imparta en sus clases, no su sustituto, y del cual el alumno deberá mejorar el contenido con sus anotaciones.

Informática Gráfica segunda edición es un libro que renueva la obra publicada en el año 2015 de título Informática Gráfica, número 107 de la colección Sapientia. Por una parte, se ha pasado a utilizar WebGL 2.0 cuya especificación apareció a principios del 2017, mucho más moderna que la versión 1.0, la cual está basada en una especificación de hace más de diez años. Por otra parte, se han realizado mejoras en todos los capítulos incluyendo cambios en prácticamente todas sus secciones, nuevas figuras, más ejemplos y ejercicios, y una nueva sección de cuestiones al final de cada capítulo.

El contenido del libro se puede dividir en tres bloques. El primer bloque lo formarían los primeros cuatro capítulos en los que se introduce la programación moderna de gráficos por computador a través de la interfaz de programación de *hardware* gráfico WebGL 2.0, el modelado y visualizado de modelos poligonales, el uso de transformaciones geométricas, transformaciones de cámara y proyección, y el problema de la visibilidad. El segundo bloque se centra principalmente en introducir técnicas para mejorar la calidad visual de la imagen sintética y lo formarían los capítulos 5 a 7. En este bloque se incluye, por ejemplo, el modelo de iluminación de Phong, modelos de sombreado, diversas técnicas para la aplicación de texturas e implementación de texturas procedurales básicas. El tercer y último bloque del libro lo forman los capítulos 8 a 10 que principalmente presentan técnicas avanzadas de realismo visual, como transparencia, espejos y sombras, métodos relacionados con el desarrollo de aplicaciones gráficas como, por ejemplo, mecanismos de interacción y animación con *shaders*, y métodos de procesamiento de imagen como la corrección *gamma* o filtros de convolución.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Recursos en línea

Se ha creado el sitio web `http://cphoto.uji.es/grafica2` como apoyo a este material, donde el lector puede descargar los programas de ejemplo que se incluyen en los diferentes capítulos.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 1

Introducción a WebGL

Índice

1.1. Antecedentes **22**
1.2. Prueba de WebGL **23**
1.3. El mínimo programa **24**
1.4. El primer triángulo **28**
1.4.1. El *pipeline* 31
1.4.2. GLSL 31

WebGL es una interfaz de programación de aplicaciones (API) para generar imágenes por ordenador en páginas web. Permite desarrollar aplicaciones interactivas que producen imágenes en color de alta calidad formadas por objetos tridimensionales (véase figura 1.1). Además, WebGL solo requiere de un navegador que lo soporte, por lo que es independiente tanto del sistema operativo como del sistema gráfico de ventanas. En este capítulo se introduce la programación con WebGL a través de un pequeño programa y se presenta el lenguaje GLSL para la programación de *shaders*.



Figura 1.1: Ejemplo de escena tridimensional dibujada con WebGL

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

1.1. Antecedentes

OpenGL se presentó en 1992. Su predecesor fue Iris GL, un API diseñado y soportado por la empresa Silicon Graphics. Desde entonces, la OpenGL Architecture Review Board (ARB) conduce la evolución de OpenGL, controlando la especificación y los tests de conformidad.

En sus orígenes, OpenGL se basó en un *pipeline* configurable de funcionamiento fijo. El usuario podía especificar algunos parámetros, pero el funcionamiento y el orden de procesamiento era siempre el mismo. Con el paso del tiempo, los fabricantes de *hardware* gráfico necesitaron dotarla de mayor funcionalidad que la inicialmente concebida. Así, se creó un mecanismo para definir extensiones que, por un lado, permitía a los fabricantes proporcionar *hardware* gráfico con mayores posibilidades, al mismo tiempo que ofrecían la capacidad de no realizar siempre el mismo *pipeline* de funcionalidad fija.

En el año 2004 aparece OpenGL 2.0, el cual incluiría el OpenGL Shading Language, GLSL 1.1, e iba a permitir a los programadores la posibilidad de escribir un código que fuese ejecutado por el procesador gráfico. Para entonces, las principales empresas fabricantes de *hardware* gráfico ya ofrecían procesadores gráficos programables. A estos programas se les denominó *shaders* y permitieron incrementar las prestaciones y el rendimiento de los sistemas gráficos de manera espectacular, al generar además una amplia gama de efectos: iluminación más realista, fenómenos naturales, texturas procedurales, procesamiento de imágenes, efectos de animación, etc. (véase figura 1.2).



Figura 1.2: Ejemplos de objetos dibujados mediante *shaders*

Dos años más tarde, el consorcio ARB pasó a ser parte del grupo Khronos. [1] Entre sus miembros activos, promotores y contribuidores se encuentran empresas de prestigio internacional como AMD, Apple, arm, Epic Games, Google, Huawei, Qualcomm, Nvidia, Intel, Nokia, etc. Es en el año 2008 cuando OpenGL, con la aparición de OpenGL 3.0 y GLSL 1.3, adopta el modelo de obsolescencia, aunque

[1] http://www.khronos.org/

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

manteniendo compatibilidad con las versiones anteriores. Sin embargo, en el año 2009, con las veriones de OpenGL 3.1 y GLSL 1.4, es cuando probablemente se realiza el cambio más significativo; el *pipeline* de funcionalidad fija y sus funciones asociadas son eliminadas, aunque disponibles aún a través de extensiones que están soportadas por la mayor parte de las implementaciones.

OpenGL ES 1.0 aparece en el 2003. Se trata de la primera versión de OpenGL para sistemas empotrados incluyendo teléfonos móviles, tabletas, consolas, vehículos, etc. Se basó en la especificación inicial de OpenGL. La versión 2.0 creada a partir de la especificación de OpenGL 2.0 aparece en el 2007, eliminando parte de la funcionalidad fija y permitiendo la programación con *shaders*. Esta versión, soportada en sistemas Android desde la versión 2 e IOS desde la versión 5, es la base para la especificación de WebGL 1.0 que aparece en el 2011 y que en la actualidad está ampliamente soportada por los navegadores Safari, Chrome, Firefox, Opera e Internet Explorer, entre otros. Derivada de OpenGL 3.3, la primera especificación de OpenGL ES 3.0 se hace pública en el 2013 y está soportada en sistemas Android desde la versión 4.3 e IOS desde su versión 7. A su vez, esta nueva versión es la base para la especificación de WebGL 2.0 que ve finalmente la luz en el año 2017.

1.2. Prueba de WebGL

Averiguar si se dispone de soporte para WebGL es muy simple. Abre un navegador y accede a cualquiera de las muchas páginas que informan de si el navegador soporta o no WebGL. Por ejemplo, la página `http://get.webgl.org` es una de ellas. Si funciona, se mostrará una página con el dibujo de un cubo en alambre dando vueltas sobre sí mismo como el que aparece en la figura 1.3.

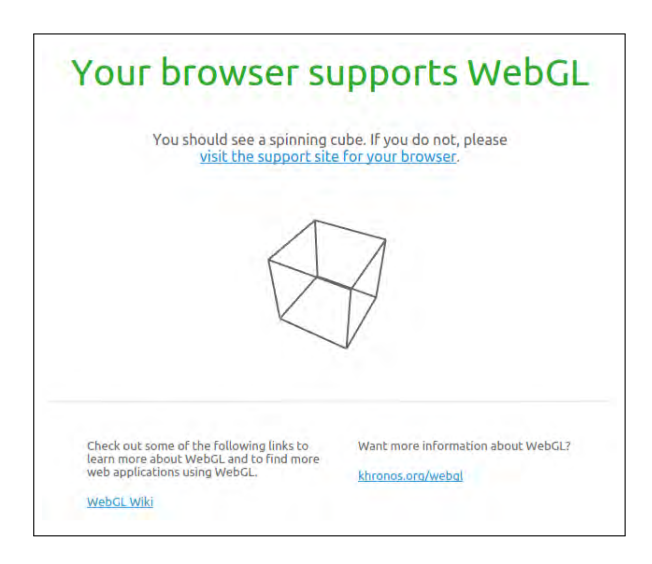


Figura 1.3: Resultado con éxito del test de soporte de WebGL en un navegador (`http://get.webgl.org`)

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

▶ **1.1** Comprueba la disponibilidad de WebGL en los diferentes navegadores que tengas instalados en tu equipo. Si tienes varios sistemas operativos repite las pruebas en cada uno de ellos. Si tienes dispositivos móviles, teléfonos o tabletas a tu alcance, prueba también el soporte con los diferentes navegadores. Después de las distintas pruebas:

- ¿Cuál es tu opinión respecto al estado de soporte de WebGL en los diferentes navegadores?
- ¿Crees que es suficiente o que por contra tendremos que esperar aún más a que aparezcan nuevas versiones de los navegadores?
- ¿Piensas que lo que desarrolles en WebGL vas a a tener que probarlo en cada navegador y sistema con el fin de comprobar, no solo su funcionamiento, sino también si se obtiene o no el mismo resultado?

▶ **1.2** Accede a la siguiente web: `http://webglreport.com/`. Obtendrás una página con un contenido similar al que se muestra en la figura 1.4. Aunque algunas características te resulten incomprensibles, trata de contestar a las siguientes preguntas:

- ¿Cuál es la versión de WebGL que soporta tu navegador?
- ¿Cuántos bits se utilizan en el *framebuffer* para codificar el color de un píxel?

▶ **1.3** Si realizas una búsqueda en internet encontrarás bastantes páginas que ofrecen una selección de ejemplos y donde también los desarrolladores pueden colgar sus propios trabajos. Algunos ejemplos son:

- Chrome Experiments[2]
- 22 Experimental WebGL Demo Examples[3]
- WebGL Samples[4]
- Plus 360 Degrees[5]

1.3. El mínimo programa

Incluir una aplicación WebGL en una página web requiere de dos pasos previos: crear un canvas y obtener un contexto WebGL. Como ya sabes, HTML es un lenguaje estandard que se utiliza para el desarrollo de páginas y aplicaciones web. La versión HTML5 incluye entre sus nuevos elementos el denominado canvas. Se trata de un elemento rectangular que establece el área de la página web donde se visualizará la aplicación WebGL. El listado 1.1 muestra el código HTML que crea un canvas de tamaño 800×600.

[2] `http://www.chromeexperiments.com/`

[3] `http://www.awwwards.com/22-experimental-webgl-demo-examples.html`

[4] `http://webglsamples.org/`

[5] `http://www.plus360degrees.com/`

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

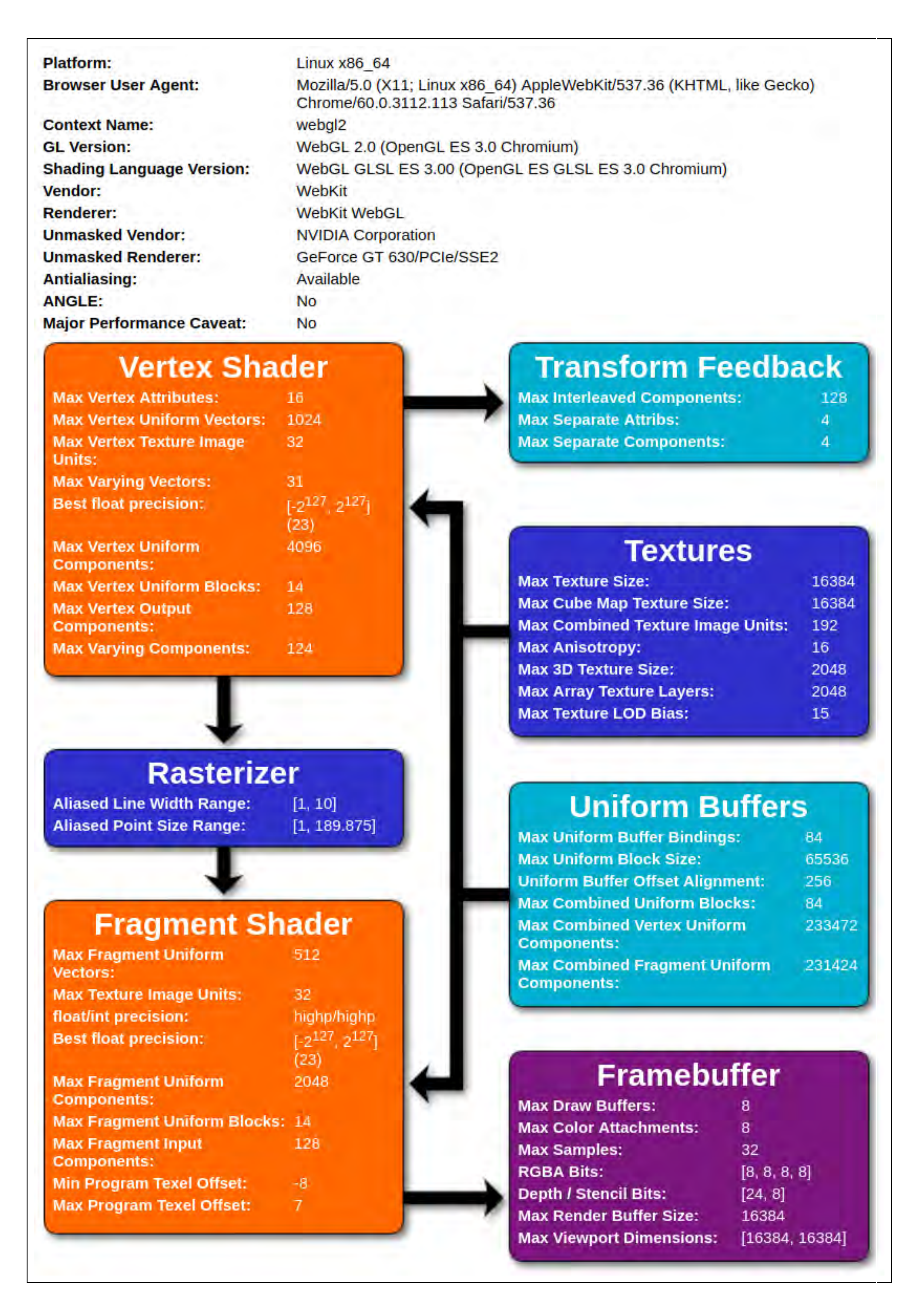


Figura 1.4: Resultado de la página `http://webglreport.org` que proporciona información sobre algunas propiedades relativas al soporte de WebGL 2.0 en el navegador.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

► **1.4** Examina el listado 1.1, utiliza un editor para escribirlo y guárdalo con el nombre *canvas.html*. Ahora ábrelo con el navegador que hayas seleccionado para trabajar. Prueba a cambiar algunos de los parámetros como el tamaño, el tipo de borde, o su color. Realmente, que el marco sea visible no es necesario, pero de momento facilita ver claramente cuál es el área de dibujo establecida.

Listado 1.1: Ejemplo de creación de un canvas con HTML5

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <title> Canvas </title>
    <style type="text/css">
      canvas {border: 1px solid black;}
    </style>
  </head>

  <body>
    <canvas id="myCanvas" width="800" height="600">
      El Navegador no soporta HTML5
    </canvas>
  </body>

</html>
```

También es el propio canvas el que nos va a proporcionar un contexto WebGL, objeto JAVASCRIPT a través del cual se accede a toda la funcionalidad. Siempre crearemos primero el canvas y entonces obtendremos el contexto WebGL. Observa el código del listado 1.2 que trata de obtener un contexto WebGL 2.0 e informa de su disponibilidad.

Ejercicios

► **1.5** Examina el listado 1.2, utiliza un editor para escribirlo y guárdalo como *contexto.js*. Ahora recupera *canvas.html* y añade el *script* justo antes de cerrar el cuerpo de la página web (etiqueta `</body>`):

- `<script src= ”contexto.js” >< /script>`

Guárdalo y refresca la página en el navegador. Comprueba el resultado, ¿tienes soporte para WebGL 2.0?

Observa ahora la nueva función *initWebGL* del listado 1.3. Esta función especifica un color de borrado, o color de fondo, utilizando el método *clearColor*, y ordena que borre el contenido del canvas con la orden *clear* y el parámetro

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

COLOR_BUFFER_BIT. Ambas instrucciones son ya órdenes de WebGL. Cabe señalar que aunque este programa contiene la mínima expresión de código que utiliza WebGL, la estructura habitual de un programa que utilice WebGL se corresponde con el que se muestra más adelante en el listado 2.4.

Listado 1.2: Código JAVASCRIPT para obtener un contexto WebGL 2.0

```
function getWebGLContext() {

  var canvas = document.getElementById("myCanvas");

  try {
    return canvas.getContext("webgl2");
  }
    catch(e) {
  }

  return null;

}

function initWebGL() {

  var gl = getWebGLContext();

  if (!gl) {
      alert("WebGL 2.0 no está disponible");
  } else {
      alert("WebGL 2.0 disponible");
  }
}

initWebGL();
```

Listado 1.3: Código que utiliza órdenes de WebGL

```
function initWebGL() {

  var gl = getWebGLContext();

  if (!gl) {
    alert("WebGL 2.0 no está disponible");
    return;
  }

  //  especifica en RGBA el color de fondo (4 valores entre 0 y 1)
  gl.clearColor(1.0,0.0,0.0,1.0);

  // rellena el buffer de color utilizando el color
  // especificado con la orden clearColor
  gl.clear(gl.COLOR_BUFFER_BIT);

}
```


ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

► **1.6** Ejecuta el programa *c01/minimoPrograma.html* que implementa la función *initWebGL* del listado 1.3. Consulta en la guía de programación de WebGL las órdenes *clear* y *clearColor* y contesta a las siguientes cuestiones:

- ¿Qué has de cambiar para que el color de fondo sea amarillo?
- ¿Qué ocurre si intercambias el orden de *clear* y *clearColor*? ¿Por qué?

1.4. El primer triángulo

El listado 1.4 muestra la nueva función *initShader* cuyo objetivo es compilar y enlazar un *shader*. Pero, ¿qué es un *shader*? Un *shader* no es más que un programa que se ejecuta en la GPU. En la actualidad, una GPU puede llegar a contener hasta cinco tipos de procesadores: de vértices, de control de teselación, de evaluación de teselación, de geometría y de fragmentos; por lo que también decimos que hay cinco tipos de *shaders*, uno por cada tipo de procesador. Sin embargo, WebGL 2.0 solo soporta dos tipos de *shaders*, el de vértices y el de fragmentos, de modo que únicamente podremos escribir *shaders* de vértices y *shaders* de fragmentos.

Un *shader*, antes de poder ser ejecutado en una GPU, debe ser compilado y enlazado. El compilador está integrado en el propio *driver* de OpenGL instalado en tu máquina (ordenador, teléfono, tableta, etc.). Esto implica que la aplicación en tiempo de ejecución será quien envíe el código fuente del *shader* al *driver* para que sea compilado y enlazado, creando un ejecutable que puede ser instalado en la GPU. Tres son los pasos a realizar:

1. Crear y compilar los objetos *shader*.
2. Crear un programa y añadirle los objetos compilados.
3. Enlazar el programa creando un ejecutable.

Ejercicios

► **1.7** El listado 1.4 muestra un ejemplo de todo el proceso. Observa detenidamente la función *initShader* e identifica en el código cada una de las tres etapas. Consulta la especificación de WebGL para conocer más a fondo cada una de las órdenes que aparecen en el ejemplo. Señalar que para que el programa ejecutable sea instalado en los procesadores correspondientes, es necesario indicarlo con la orden *glUseProgram*, que como parámetro debe recibir el identificador del programa que se desea utilizar. La carga de un ejecutable siempre supone el desalojo del que hubiera con anterioridad.

► **1.8** Edita el fichero *c01/miPrimerTrianguloConWebGL.js*. Observa cómo queda incluida la función *initShader* dentro de la aplicación y en qué momento se le llama desde la función *initWebGL*.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 1.4: Compilación y enlazado de un *shader*

```
function initShader() {

  // paso 1
  var vertexShader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(vertexShader,
           document.getElementById('myVertexShader').text);
  gl.compileShader(vertexShader);

  var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fragmentShader,
           document.getElementById('myFragmentShader').text);
  gl.compileShader(fragmentShader);

  // paso 2
  program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);

  // paso 3
  gl.linkProgram(program);

  gl.useProgram(program);

  program.vertexPositionAttribute = gl.getAttribLocation(
      program, "VertexPosition");
  gl.enableVertexAttribArray(program.vertexPositionAttribute);
}
```

El listado 1.5 muestra un ejemplo de *shader*, el más simple posible. Los *scripts* identificados como *myVertexShader* y *myFragmentShader* contienen los códigos fuente del *shader* de vértices y del *shader* de fragmentos respectivamente. Estos *scripts* se deben incluir en el fichero HTML.

Cuando desde la aplicación se ordene dibujar un modelo poligonal, cada vértice producirá la ejecución del *shader* de vértices, el cual a su vez produce como salida la posición del vértice que se almacena en la variable predefinida *gl_Position*. El resultado de procesar cada vértice atraviesa el *pipeline*, los vértices se agrupan dependiendo del tipo de primitiva a dibujar, y en la etapa de conversión al *raster* la posición del vértice (y también de sus atributos en el caso de haberlos) es interpolada, generando los fragmentos y produciendo, cada uno de ellos, la ejecución del *shader* de fragmentos. El propósito de este último *shader* es determinar el color definitivo del fragmento. Siguiendo con el ejemplo, todos los fragmentos son puestos a color verde (especificado en formato RGBA) utilizando la variable *fragmentColor*.

Ejercicios

► **1.9** Carga en el navegador el fichero *c01/miPrimerTrianguloConWebGL.html* y comprueba que obtienes una salida similar a la que se muestra en la figura 1.5.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

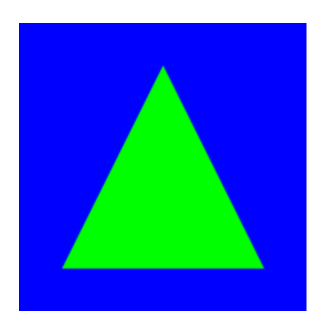


Figura 1.5: El primer triángulo con WebGL

Listado 1.5: Ejemplo básico de *shader* de vértices y *shader* de fragmentos

```
<script id="myVertexShader"
      type="x-shader/x-vertex">#version 300 es

  // Shader de vértices

  // Declaración del atributo posición
  in vec3 VertexPosition;

  void main()  {

    // se asigna la posición del vértice a
    // la variable predefinida gl_Position
    gl_Position = vec4(VertexPosition, 1.0);

  }
</script>

<script id="myFragmentShader"
      type="x-shader/x-fragment">#version 300 es

  // Shader de fragmentos

  precision mediump float;

  out vec4 fragmentColor;

  void main() {

    // se asigna el color verde a cada fragmento
    fragmentColor = vec4(0.0,1.0,0.0,1.0);

  }
</script>
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

1.4.1. El *pipeline*

El funcionamiento básico del *pipeline* se representa en el diagrama simplificado que se muestra en la figura 1.6. Las etapas de procesado del vértice y del fragmento son programables, y es el programador el responsable de escribir los *shaders* que se han de ejecutar en cada una de ellas.

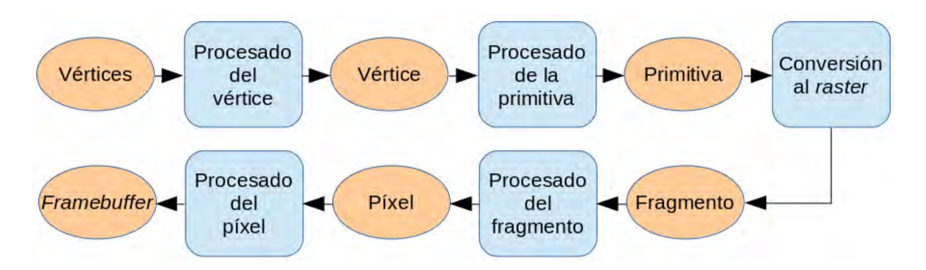


Figura 1.6: Secuencia básica de operaciones del *pipeline* de WebGL

El procesador de vértices acepta vértices como entrada, los procesa utilizando el *shader de vértices* y envía el resultado a la etapa denominada «procesado de la primitiva». En esta etapa, los vértices se reagrupan dependiendo de qué primitiva geométrica se está procesando (punto, línea o triángulo). También se realizan otras operaciones que de momento se van a omitir. La primitiva pasa por una etapa de «conversión al *raster*», que básicamente consiste en generar pequeños trozos denominados fragmentos que todos juntos cubren la superficie de la primitiva.

El procesador de fragmentos determina el color definitivo de cada fragmento utilizando el *shader de fragmentos*. El resultado se envía al *framebuffer* no sin antes atravesar algunas etapas que de momento también se omiten por cuestión de claridad.

1.4.2. GLSL

El lenguaje GLSL forma parte de WebGL y permite al programador escribir el código que desea ejecutar en los procesadores programables de la GPU. WebGL 2.0 utiliza la versión 3.0 ES de GLSL. Por este motivo los *shaders* comienzan con la directiva `#version 300 es` (véase listado 1.5).

GLSL es un lenguaje de alto nivel, parecido al C, aunque también toma prestadas algunas características del C++. Su sintaxis se basa en el ANSI C. Constantes, identificadores, operadores, expresiones y sentencias son básicamente las mismas que en C. El control de flujo con bucles, la sentencias condicionales *if-then-else* y las llamadas a funciones son idénticas al C. Pero GLSL también añade características no disponibles en C, entre otras se destacan las siguientes:

- Tipos vector: vec2, vec3, vec4
- Tipos matriz: mat2, mat3, mat4

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

- Tipos *sampler* para el acceso a texturas: sampler2D, samplerCube
- Tipos para comunicarse entre *shaders* y con la aplicación: *in*, *out*, *uniform*
- Acceso a componentes de un vector mediante: .xyzw .rgba .stpq
- Operaciones vector-matriz, por ejemplo: $vec4\ a = b * c$, siendo b de tipo $vec4$ y c de tipo $mat4$
- Variables predefinidas que almacenan estados de WebGL

GLSL también dispone de funciones propias como, por ejemplo, trigonométricas (*sin*, *cos*, *tan*, etc.), exponenciales (*pow*, *exp*, *sqrt*, etc.), comunes (*abs*, *floor*, *mod*, etc.), geométricas (*length*, *cross*, *normalize*, etc.), matriciales (*transpose*, *inverse*, etc.) y operaciones relacionales con vectores (*equal*, *lessThan*, *any*, etc). Consulta la especificación del lenguaje para conocer el listado completo. También hay características del C que no están soportadas en OpenGL, como es el uso de punteros, de los tipos: *byte*, *char*, *short*, *long int* y la conversión implícita de tipos está muy limitada. Del C++, GLSL copia la sobrecarga y el concepto de constructor.

Por último indicar que en GLSL hay que establecer la precisión numérica de las variables mediante el uso de uno de estos tres calificadores: *lowp*, *mediump*, *highp*. Estos calificadores se pueden utilizar tanto en la declaración de la variable como de forma general para todas las variables de un determinado tipo. Este último caso se realiza mediante la sentencia:

- `precision calificador_de_precisión tipo;`

Por ejemplo, en el listado 1.5 se ha utilizado el calificador *mediump* para las variables de tipo *float*. Sin embargo, en su lugar se podría haber utilizado el calificador de precisión directamente en la declaración de la correspondiente variable:

- `out mediump vec4 fragmentColor;`

En el *shader* de vértices todos los tipos tienen una precisión establecida por defecto. En el *shader* de fragmentos ocurre lo mismo a excepción del tipo *float*, para el que es necesario que el programador lo especifique en el propio *shader*. Consulta la especificación del lenguaje para conocer el calificador establecido por defecto para cada tipo así como la precisión numérica asociada a cada calificador.

Ejercicios

► **1.10** Modifica el color de relleno del triángulo del ejercicio anterior por otro que tu elijas. Recuerda que el color de cada fragmento se establece en el *shader* de fragmentos cuyo código fuente está en el fichero HTML.

► **1.11** Aunque aún no se ha tratado el dibujado de geometría, prueba a modificar las coordenadas de los vértices del triángulo definido en la variable `exampleTriangle`. Por ejemplo, trata de obtener el triángulo simétrico respecto a un eje horizontal. Prueba también a especificar coordenadas de magnitud mayor que uno, e incluso a utilizar valores distintos de cero en la componente Z. Haz todas las pruebas que se te ocurran, es la mejor manera de aprender.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Cuestiones

▶ **1.1** ¿Qué es un canvas?

▶ **1.2** ¿Qué es un contexto WebGL?

▶ **1.3** ¿Creas primero el canvas y a partir de él un contexto WebGL o al revés?

▶ **1.4** ¿En qué lenguaje se codifica la obtención de un contexto WebGL, HTML o JavaScript?

▶ **1.5** ¿Qué elemento de HTML te permite cargar un fichero que contenga código escrito en JavaScript?

▶ **1.6** Explica para qué sirve la función de WebGL `clearColor`. ¿Cómo se complementa con la función `clear`? ¿A qué hace referencia `COLOR_BUFFER_BIT`?

▶ **1.7** ¿Qué es un *shader*?

▶ **1.8** ¿Qué tipos de *shaders* tienes disponibles en WebGL 2.0?

▶ **1.9** ¿Qué es un fragmento? ¿En que tipo de *shader* se establece el color definitivo de un fragmento?

▶ **1.10** ¿Qué variable ha de almacenar el color definitivo de un fragmento?

▶ **1.11** ¿Qué variable ha de almacenar las coordenadas definitivas resultado de procesar un vértice?

▶ **1.12** En el *pipeline* de WebGL, ¿qué se realiza antes, el procesado del vértice o el del fragmento?

▶ **1.13** ¿En qué lenguaje y versión se escribe un *shader* para WebGL 2.0?

▶ **1.14** ¿Para qué sirve la orden `useProgram`?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 2

Modelado poligonal

Índice

2.1. Representación 36
2.1.1. Caras independientes 37
2.1.2. Vértices compartidos 37
2.1.3. Tiras y abanicos de triángulos 39
2.2. La normal 41
2.3. Mallas y WebGL 43
2.3.1. Preparación y dibujado 44
2.3.2. Tipos de primitivas geométricas 44
2.3.3. Variables *uniform* 50
2.3.4. Variables *out* 51

Se denomina modelo al conjunto de datos que describe un objeto y que puede ser utilizado por un sistema gráfico para ser visualizado. Hablamos de modelo poligonal cuando se utilizan polígonos para describirlo. En general, el triángulo es la primitiva más utilizada, aunque también el cuadrilátero se emplea en algunas ocasiones (véase figura 2.1).

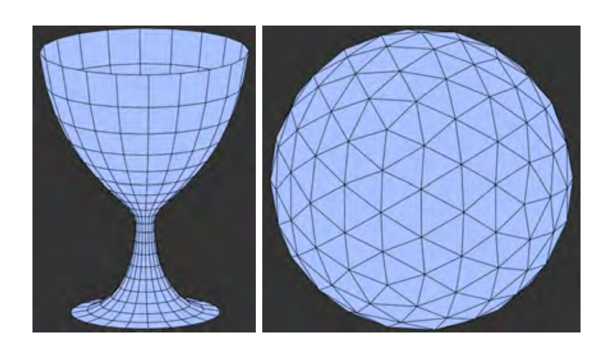


Figura 2.1: A la izquierda, objeto representado mediante cuadriláteros y a la derecha, objeto representado mediante triángulos

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

El *hardware* gráfico actual se caracteriza por su velocidad a la hora de pintar polígonos. Los fabricantes anuncian desde hace años tasas de dibujado de varios millones de polígonos por segundo. Por esto, el uso de modelos poligonales para visualizar modelos 3D en aplicaciones interactivas es prácticamente una obligación. Por contra, los modelos poligonales representan las superficies curvas de manera aproximada, como se puede observar en la figura 2.1. Sin embargo, hay métodos que permiten visualizar un modelo poligonal de modo que este sea visualmente exacto. Por ejemplo, la figura 2.2 muestra la representación poligonal del modelo de una copa en la que se puede observar que el hecho de que la superficie curva se represente mediante un conjunto de polígonos planos no impide que la observemos como si de una superficie curva se tratara.

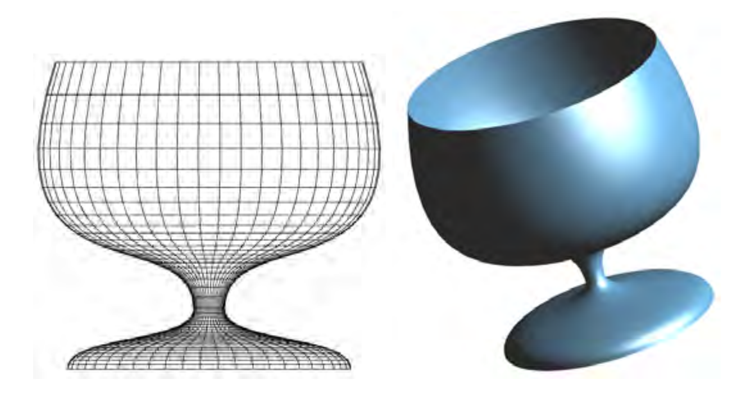


Figura 2.2: Representación poligonal de una copa y resultado de su visualización

2.1. Representación

Normalmente los modelos poligonales representan objetos donde aristas y vértices se comparten entre diferentes polígonos. A este tipo de modelos se les denomina mallas poligonales. La figura 2.3 muestra varios ejemplos (modelos cortesía del *Stanford Computer Graphics Laboratory* [1]). A la hora de definir una estructura para la representación de mallas poligonales es importante tener en cuenta esta característica para tratar de reducir el espacio de almacenamiento, el consumo de ancho de banda y el tiempo de dibujado. La tabla 2.1 muestra los datos de cada uno de los modelos de la figura 2.3.

	Bunny	*Armadillo*	*Dragon*
#Vértices	35.947	172.974	435.545
#Triángulos	69.451	345.944	871.306

Tabla 2.1: Datos de los modelos *Bunny*, *Armadillo* y *Dragon*

[1] http://graphics.stanford.edu/data/3Dscanrep/

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

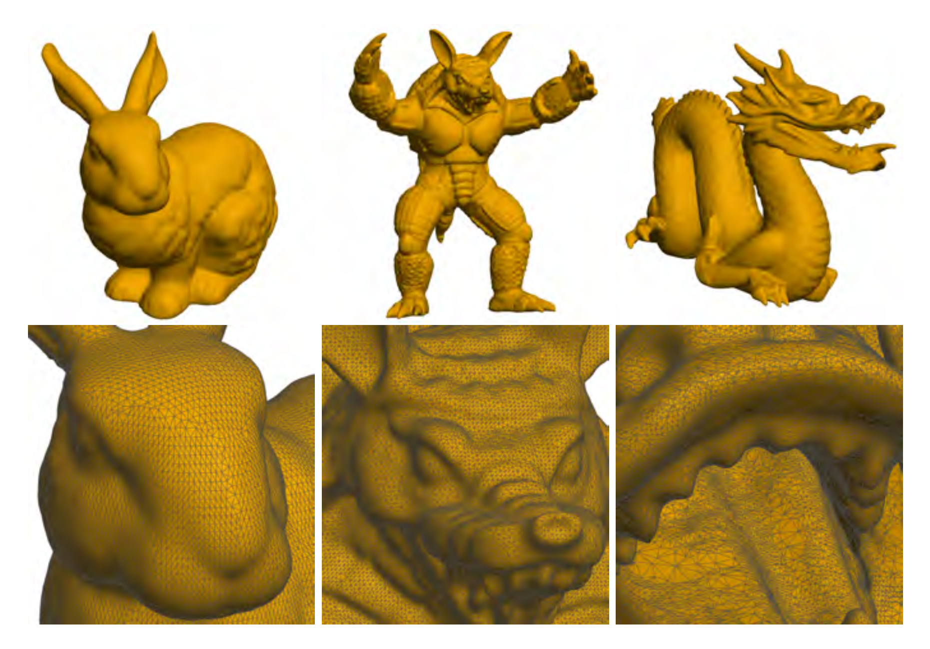


Figura 2.3: Ejemplos de mallas poligonales, de izquierda a derecha, *Bunny*, *Armadillo* y *Dragon* (modelos cortesía del *Stanford Computer Graphics Laboratory*)

2.1.1. Caras independientes

Este tipo de representación se caracteriza por almacenar cada triángulo de manera independiente, como si estos no compartieran información alguna. Su estructura de datos (véase listado 2.1) se correspondería con una única lista de triángulos, donde para cada triángulo se almacenan las coordenadas de cada uno de sus vértices, tal y como se muestra en la figura 2.4.

Listado 2.1: Estructura correspondiente a la representación de caras independientes

```
struct triángulo {

  vector3 coordenadas[3];

} Triángulos[nTriángulos];
```

2.1.2. Vértices compartidos

En este caso se separa la información de los vértices y la de los triángulos en dos listas (véase listado 2.2). De esta manera, cada vértice compartido se almacena una única vez y cada triángulo se representa mediante tres índices a la lista de vértices (véase figura 2.5).

La tabla 2.2 muestra que el uso de la estructura de vértices compartidos reduce a aproximadamente la mitad el coste de almacenamiendo que conlleva utilizar la estructura de caras independientes.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

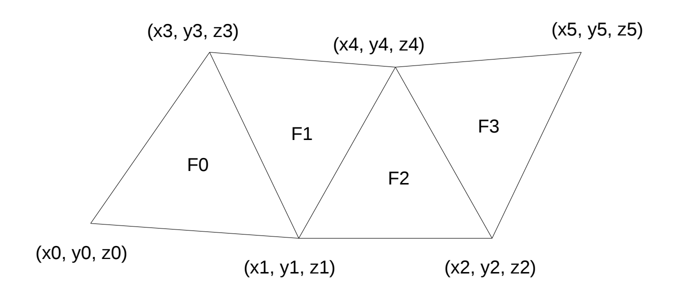


Figura 2.4: Esquema de almacenamiento de una malla poligonal mediante la estructura de caras independientes

Listado 2.2: Estructura correspondiente a la representación de vértices compartidos

```
struct vértice {

  float coordenadas[3];

} Vértices[nVértices];

struct triángulo {

  unsigned int índices[3];

} Triángulos[nTriángulos];
```

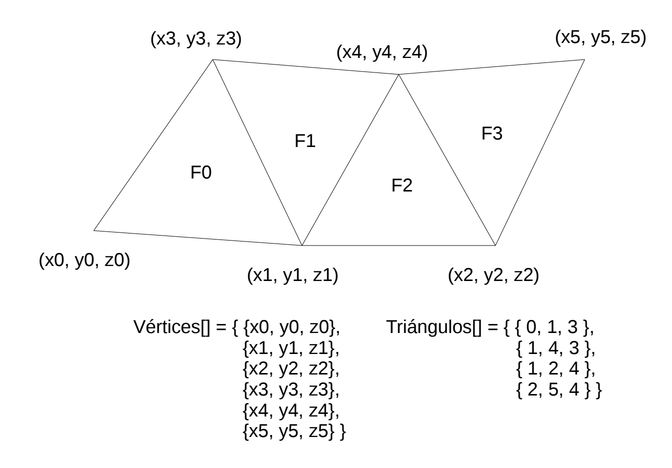


Figura 2.5: Esquema de almacenamiento de una malla poligonal mediante la estructura de vértices compartidos

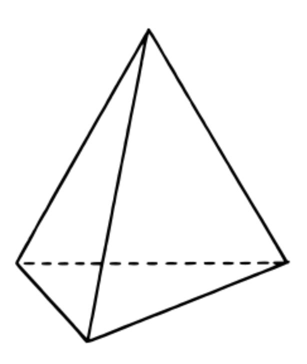
José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

	Bunny	*Armadillo*	*Dragon*
Caras independientes	2.500.236	12.453.984	31.367.016
Vértices compartidos	1.264.776	6.227.016	15.682.212

Tabla 2.2: Costes de almacenamiento en *bytes* de los modelos *Bunny*, *Armadillo* y *Dragon* asumiendo que un real de simple precisión o un entero son 4 *bytes*

Ejercicios

► **2.1** Comprueba que los costes de almacenamiento según el tipo de estructura utilizada que se muestran en la tabla 2.2 son correctos dados los datos de la tabla 2.1.
► **2.2** Obtén y compara los costes de almacenamiento en *bytes* de un tetraedro utilizando los dos tipos de representación. Asume que almacenar una coordenada o un índice cuesta 4 *bytes*.



2.1.3. Tiras y abanicos de triángulos

Estos tipos de representación tratan de aumentar las prestaciones del sistema gráfico creando grupos de triángulos que comparten vértices.

En el caso de un grupo de tipo tira de triángulos, los primeros tres vértices definen el primer triángulo. Cada nuevo vértice define un nuevo triángulo utilizando ese vértice y los dos últimos del triángulo anterior (véase figura 2.6 derecha).

En el caso de un grupo de tipo abanico de triángulos, su primer vértice representa a un vértice común a todos los triángulos del mismo grupo. De nuevo, los tres primeros vértices definen el primer triángulo, pero después cada nuevo vértice produce que se elimine el segundo vértice del triángulo anterior y se agrege el nuevo (véase figura 2.6 izquierda).

También es posible utilizar tiras de triángulos generalizadas que permiten, por ejemplo, representar mediante una sola tira la malla de la figura 2.7. Esto se consigue repitiendo vértices que producen a su vez triángulos degenerados (triángulos donde dos de sus vértices son el mismo).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

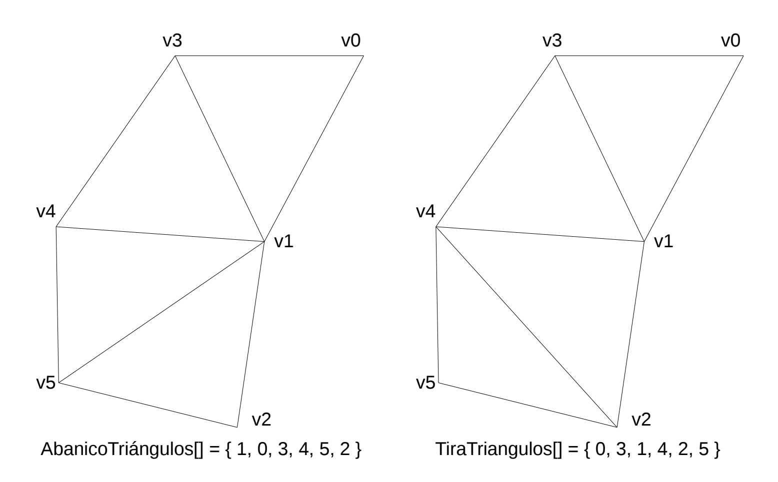


Figura 2.6: Ejemplo de abanico y tira de triángulos

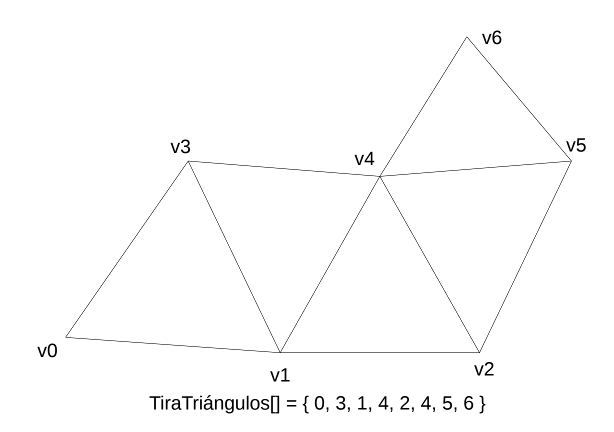


Figura 2.7: Representación de la malla mediante una tira de triángulos generalizada

La tabla 2.3 muestra para el modelo *Bunny* que el uso de la tira de triángulos en conjunto con la estructura de vértices compartidos reduce a aproximadamente un tercio el coste de almacenamiendo comparado con utilizar triángulos y la estructura de caras independientes. Estos datos se han obtenido utilizando una única tira de triángulos que representa toda la malla y está compuesta por 102.997 índices. La tabla también muestra que, en este ejemplo, el uso de tiras de triángulos ha producido un incremento del 50 % en el número de imágenes por segundo (*Fps*) independientemente del tipo de estructura utilizada.

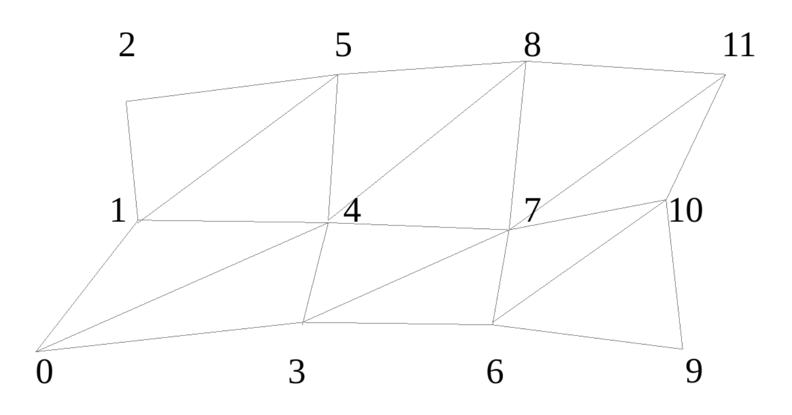
Ejercicios

▶ **2.3** Obtén la representación de un tetraedro mediante una única tira de triángulos, ¿consigues hacerlo sin utilizar triángulos degenerados? Obtén el coste de almacenamiento utilizando la estructura de vértices compartidos, ¿has reducido el coste al utilizar la tira de triángulos en lugar de triángulos?

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

► **2.4** Obtén la representación de la siguiente malla mediante una única tira de triángulos y contesta, ¿cuántos triángulos degenerados has introducido?



Primitiva	**Estructura**	***Bytes***	***Fps***
Triángulo	Caras independientes	2.500.236	20
	Vértices compartidos	1.264.776	20
Tira de triángulos	Caras independientes	1.235.964	30
	Vértices compartidos	843.352	30

Tabla 2.3: Costes de almacenamiento en *bytes* y tasas de imágenes por segundo (*Fps*, del inglés *frames per second*) del modelo *Bunny* utilizando diferentes estructuras y primitivas

2.2. La normal

Un modelo poligonal, además de vértices y caras, suele almacenar otra información a modo de atributos como el color, la normal o las coordenadas de textura. Estos atributos son necesarios para mejorar el realismo visual. Por ejemplo, en la figura 2.8 se muestra el resultado de la visualización del modelo poligonal de una tetera obtenido gracias a que, además de la geometría, se ha proporcionado la normal de la superficie para cada vértice.

La normal es un vector perpendicular a la superficie en un punto. Si la superficie es plana, la normal es la misma para todos los puntos de la superficie. Para un triángulo es fácil obtenerla realizando el producto vectorial de dos de los vectores directores de sus aristas. Ya que el producto vectorial no es conmutativo, es muy importante establecer cómo se va a realizar el cálculo y también que los vértices que forman las caras se especifiquen siempre en el mismo orden, para así obtener todas las normales de manera consistente.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

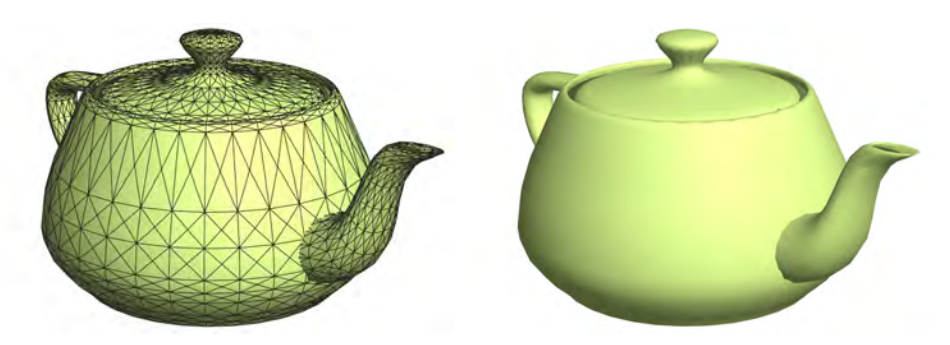


Figura 2.8: Visualización del modelo poligonal de una tetera. En la imagen de la izquierda se pueden obervar los polígonos utilizados para representarla

Ejercicios

► **2.5** Averigua cómo saber si todas las caras de una malla de triángulos están definidas en el mismo orden (horario o antihorario), y cómo arreglarlo en el caso de encontrar caras definidas en sentidos distintos.

► **2.6** Averigua cómo saber si una malla de triángulos es cerrada o si, por contra, presenta algún agujero como, por ejemplo, ocurre en la copa de la figura 2.2.

► **2.7** Observa la siguiente descripción poligonal de un objeto. Las líneas que comienzan por v se corresponden con los vértices e indican sus coordenadas. El primero se referencia con el número 1 y los demás se enumeran de forma consecutiva. Las líneas que comienzan por f se corresponden con las caras e indican qué vertices lo forman.

```
v 0 0 0
v 0 0 1
v 1 0 1
v 1 0 0
v 0 1 0
v 0 1 1
v 1 1 1
v 1 1 0
f 1 3 2
f 1 4 3
f 1 2 5
f 2 6 5
f 3 2 6
f 3 6 7
f 3 4 7
f 4 8 7
f 4 1 8
f 1 5 8
```

- Dibújalo en papel, ¿qué objeto representa?
- ¿Están todas sus caras definidas en el mismo orden?
- ¿En qué sentido están definidas, horario o antihorario?

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

- Obtén las tiras de triángulos que representan el objeto, ¿puedes conseguirlo con solo una tira?
- Calcula la normal para cada vértice, ¿qué problema encuentras?

2.3. Mallas y WebGL

Habitualmente se suele asociar el concepto de vértice con las coordenadas que definen la posición de un punto en el espacio. En WebGL, el concepto de vértice es más general, entendiéndose como una agrupación de datos a los que se denominan atributos. Por ejemplo, la posición y la normal son dos de los atributos más comunes de un vértice. En cuanto al tipo de los atributos, estos pueden ser reales, enteros, vectores, etc. Y respecto al número máximo de atributos que un vértice puede tener en WebGL 2.0, se puede consultar con la orden `gl.getParameter(gl.MAX_VERTEX_ATTRIBS)` que habitualmente retorna 16 (que es el valor mínimo establecido por el estándar).

WebGL no proporciona mecanismos para describir o modelar objetos geométricos complejos, sino que proporciona mecanismos para especificar cómo dichos objetos deben ser dibujados. Es responsabilidad del programador definir las estructuras de datos adecuadas para almacenar la descripción del objeto. Sin embargo, como WebGL requiere que la información que vaya a visualizarse se disponga en vectores, lo habitual es utilizar también vectores para almacenar los vértices (sus atributos) y utilizar índices a dichos vectores para definir las primitivas geométricas (véase listado 2.3).

Listado 2.3: Modelo de un cubo definido con triángulos dispuesto para ser utilizado con WebGL

```
var exampleCube = {

    "vertices" : [-0.5, -0.5,  0.5,
                   0.5, -0.5,  0.5,
                   0.5,  0.5,  0.5,
                  -0.5,  0.5,  0.5,
                  -0.5, -0.5, -0.5,
                   0.5, -0.5, -0.5,
                   0.5,  0.5, -0.5,
                  -0.5,  0.5, -0.5],

    "indices" : [ 0, 1, 2, 0, 2, 3,
                  1, 5, 6, 1, 6, 2,
                  3, 2, 6, 3, 6, 7,
                  5, 4, 7, 5, 7, 6,
                  4, 0, 3, 4, 3, 7,
                  4, 5, 1, 4, 1, 0]

};
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

2.3.1. Preparación y dibujado

En primer lugar, el modelo poligonal se ha de almacenar en *buffer objects*. Un *buffer object* no es más que una porción de memoria reservada dinámicamente y controlada por el propio procesador gráfico. Siguiendo con el ejemplo del listado 2.3 se necesitan dos *buffers*, uno para el vector de vértices y otro para el de índices. Después hay que asignar a cada *buffer* sus datos correspondientes. El listado 2.4 recoge estas operaciones en la función *initBuffers*, examínalo y acude a la especificación del lenguaje para conocer más detalles de las funciones utilizadas.

El listado 2.5 muestra el código HTML que incluye dos *scripts*, uno para cada tipo de *shader*, el de vértices y el de fragmentos. Observa que en el *shader* de vértices se define un único atributo de posición para cada vértice (utilizando la palabra clave `in`). El segundo paso consiste en obtener los índices de las variables del *shader* que representan los atributos de los vértices (que de momento es solo la posición) y habilitar el vector correspondiente. Estas operaciones se muestran en el listado 2.4 y se han incluido al final de la función *initShaders*, que es la encargada de compilar, enlazar y crear un ejecutable del *shader*.

Una vez que el modelo ya se encuentra almacenado en la memoria controlada por la GPU, el *shader* ya está compilado y enlazado, y los índices de los atributos de los vértices ya se han obtenido, solo queda el último paso: su visualización. Primero, hay que indicar los *buffers* que contienen los vértices y los índices correspondientes al modelo que se va a visualizar. También hay que especificar cómo se encuentran dispuestos cada uno de los atributos de los vértices en el *buffer* correspondiente. Después ya se puede ordenar el dibujado, indicando tipo de primitiva y número de elementos. Los vértices se procesarán de manera independiente, pero siempre en el orden en el que son enviados al procesador gráfico. Estas operaciones se muestran en el listado 2.4, en la función *draw*. De nuevo, acude a la especificación del lenguaje para conocer más detalles de las órdenes utilizadas.

2.3.2. Tipos de primitivas geométricas

Las primitivas básicas de dibujo en WebGL son el punto, el segmento de línea y el triángulo. Cada primitiva se define especificando sus respectivos vértices, y estas se agrupan a su vez para definir objetos de mayor complejidad. En WebGL, la primitiva geométrica se utiliza para especificar cómo han de ser agrupados los vértices tras ser operados en el procesador de vértices y así poder ser visualizada. Son las siguientes:

- Dibujo de puntos:
  - gl.POINTS
- Dibujo de líneas:
  - Segmentos independientes: gl.LINES

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

- Secuencia o tira de segmentos: gl.LINE_STRIP
- Secuencia cerrada de segmentos: gl.LINE_LOOP

- Triángulos:

  - Triángulos independientes: gl.TRIANGLES
  - Tira de triángulos: gl.TRIANGLE_STRIP
  - Abanico de triángulos: gl.TRIANGLE_FAN

Ejercicios

▶ **2.8** ¿Cuántos triángulos crees que se pintarían con la siguiente orden? (ten en cuenta que al tratarse de una tira pueden haber triángulos degenerados)

- `gl.drawElements(gl.TRIANGLE_STRIP, 30, gl.UNSIGNED_SHORT, 12)`

▶ **2.9** Abre con el navegador el archivo *c02/dibuja.html*, que incluye el código del listado 2.5, y carga a su vez a *c02/dibuja.js*, que contiene el código del listado 2.4. ¿Qué objeto se muestra?

▶ **2.10** Sustituye el modelo del triángulo en *dibuja.js* por el modelo del pentágono que se muestra a cotinuación:

```
var examplePentagon = {

  "vertices" : [ 0.0,   0.9, 0.0,
                −0.95, 0.2, 0.0,
                −0.6, −0.9, 0.0,
                 0.6, −0.9, 0.0,
                 0.95, 0.2, 0.0],

  "indices" : [ 0, 1, 2, 3, 4]

};
```

Ahora dibújalo utilizando puntos primero y después líneas:

- Puntos: `gl.drawElements(gl.POINTS, 5, gl.UNSIGNED_SHORT, 0);`. Si no eres capaz de distinguir los puntos añade en la función *main* del *shader* de vértices: `gl_PointSize = 5.0;`
- Líneas: `gl.drawElements(gl.LINE_LOOP, 5, gl.UNSIGNED_SHORT,0);`

▶ **2.11** Realiza las modificaciones necesarias para pintar el pentágono del ejercicio anterior como triángulos independientes. De esta manera pasarás a verlo relleno (en lugar de solo sus aristas). Piensa primero y haz los cambios después, y recuerda que

1. Aunque los vértices son los mismos, tendrás que modificar el vector de índices para ahora especficar los triángulos.
2. Los valores de los parámetros de la orden `drawElements` van a cambiar.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **2.12** Continúa utilizando los mismos vértices del modelo del pentágono y realiza las modificaciones necesarias para obtener una estrella como la de la figura 2.9. Prueba también a cambiar el grosor del trazo mediante la orden `gl.lineWidth(5.0)` justo antes de ordenar el dibujado de la geometría (esto no funciona en *Windows*).

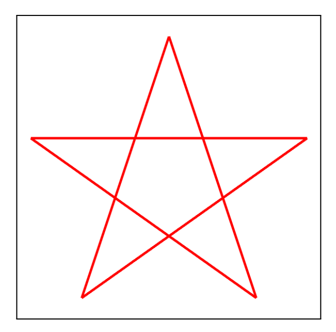


Figura 2.9: Estrella dibujada a partir de los vértices de un pentágono

▶ **2.13** Utiliza ahora la descripción de los vértices que figura a continuación. Crea una función que se llame *drawGeometryLines* y que utilizando dichos vértices dibuje la estrella con líneas de manera que se obtenga el resultado de la figura 2.10 (imagen de la izquierda). Crea otra función que se llame *drawGeometryTriangles* y que dibuje la estrella mediante triángulos independientes de manera que se obtenga el resultado que se muestra en la figura 2.10 (imagen de la derecha).

```
"vertices" : [0.0,    0.9,   0.0,
             -0.95,   0.2,   0.0,
             -0.6,   -0.9,   0.0,
              0.6,   -0.9,   0.0,
              0.95,   0.2,   0.0,
              0.0,   -0.48,  0.0,
              0.37,  -0.22,  0.0,
              0.23,   0.2,   0.0,
             -0.23,   0.2,   0.0,
             -0.37,  -0.22,  0.0],
```

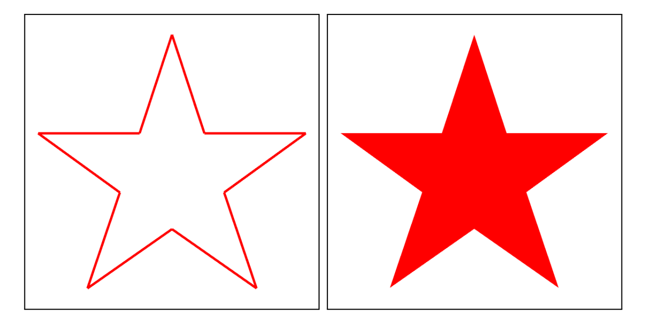


Figura 2.10: Estrella dibujada con líneas (izquierda) y con triángulos (derecha)

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 2.4: Código mínimo que se corresponde con la estructura básica de un programa que utiliza WebGL (disponible en *c02/dibuja.js*)

```
var gl, program;

var exampleTriangle = {

  "vertices" : [-0.7, -0.7, 0.0,
                 0.7, -0.7, 0.0,
                 0.0,  0.7, 0.0],
  "indices" : [ 0, 1, 2]
};

function getWebGLContext() {

  var canvas = document.getElementById("myCanvas");
  try {
    return canvas.getContext("webgl2");
  }
    catch(e) {
  }
  return null;

}

function initShaders() {

  var vertexShader = gl.createShader(gl.VERTEX_SHADER);
  gl.shaderSource(vertexShader,
      document.getElementById("myVertexShader").text);
  gl.compileShader(vertexShader);

  var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
  gl.shaderSource(fragmentShader,
      document.getElementById("myFragmentShader").text);
  gl.compileShader(fragmentShader);

  program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);

  gl.linkProgram(program);

  gl.useProgram(program);

  // Obtener la referencia del atributo posición
  program.vertexPositionAttribute =
       gl.getAttribLocation( program, "VertexPosition");

  // Habilitar el atributo posición
  gl.enableVertexAttribArray(program.vertexPositionAttribute);

}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

```
function initBuffers(model) {

  // Buffer de vértices
  model.idBufferVertices = gl.createBuffer ();
  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.bufferData (gl.ARRAY_BUFFER,
      new Float32Array(model.vertices), gl.STATIC_DRAW);

  // Buffer de índices
  model.idBufferIndices = gl.createBuffer ();
  gl.bindBuffer (gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  gl.bufferData (gl.ELEMENT_ARRAY_BUFFER,
      new Uint16Array(model.indices), gl.STATIC_DRAW);
}

function initRendering() {
  gl.clearColor(0.15,0.15,0.15,1.0);
}

function draw(model) {

  gl.bindBuffer(gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer(program.vertexPositionAttribute, 3,
      gl.FLOAT, false, 0, 0);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  gl.drawElements(gl.TRIANGLES, 3, gl.UNSIGNED_SHORT, 0);
}

function drawScene() {

  gl.clear(gl.COLOR_BUFFER_BIT);
  draw(exampleTriangle);

}

function initWebGL() {

  gl = getWebGLContext();
  if (!gl) {
    alert("WebGL 2.0 no está disponible");
    return;
  }

  initShaders();
  initBuffers(exampleTriangle);
  initRendering();

  requestAnimationFrame(drawScene);

}

initWebGL();
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 2.5: Código HTML que incluye un canvas y los dos *shaders* básicos (disponible en *c02/dibuja.html*)

```
<!DOCTYPE html>
<html>

  <head>
    <meta charset="utf-8">
    <title> Dibuja </title>
    <style type="text/css">
      canvas {border: 1px solid black;}
    </style>

    <script id="myVertexShader"
          type="x-shader/x-vertex">#version 300 es

      in vec3 VertexPosition;

      void main()  {

        gl_Position = vec4(VertexPosition, 1.0);

      }

    </script>

    <script id="myFragmentShader"
          type="x-shader/x-fragment">#version 300 es

      precision mediump float;
      out vec4 fragmentColor;

      void main() {

        fragmentColor = vec4(0.0,1.0,0.0,1.0);

      }

    </script>
  </head>

  <body>

    <canvas id="myCanvas" width="600" height="600">
      El Navegador no soporta HTML5
    </canvas>

    <script src = "dibuja.js"></script>

  </body>

</html>
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

2.3.3. Variables *uniform*

Imagina que, por ejemplo, quieres que el color de la estrella de los ejercicios anteriores no sea fijo sino que pueda cambiarse de forma dinámica. Para conseguirlo es necesario que el color que figura en el *shader* de fragmentos sea una variable, tal y como se muestra en el listado 2.6.

Listado 2.6: Ejemplo de variable *uniform* en el *shader* de fragmentos

```
#version 300 es

precision mediump float;

uniform vec4 myColor;
out     vec4 fragmentColor;

void main() {

  fragmentColor = myColor;

}
```

La variable *myColor* es de tipo *uniform* porque su valor será constante para todos los fragmentos que reciba procedentes de procesar la estrella, pero se puede hacer que sea diferente para cada estrella cambiando su valor antes de ordenar su dibujado. Observa el fragmento de código del listado 2.7. Esas líneas son las encargadas de obtener el índice de la variable *myColor* en el *shader* y de especificar su valor. Mientras que la primera línea solo es necesario ejecutarla una vez, la segunda habrá que utilizarla cada vez que se necesite asignar un nuevo valor a la variable *myColor*. Así, la primera línea se podría añadir al final de la función *initShaders*, mientras que la segunda línea habrá que añadirla justo antes de ordenar el dibujado del modelo.

Listado 2.7: Obtiene la referencia y establece el valor de la variable *uniform* `myColor`

```
var idMyColor = gl.getUniformLocation(program, "myColor");
gl.uniform4f(idMyColor, 1.0, 0.0, 1.0, 1.0);
```

Ejercicios

► **2.14** Dibuja dos estrellas de colores distintos, una para pintar el interior y la otra para pintar el borde (véase figura 2.11). Elije colores que contrasten entre sí. Ten en cuenta también que el orden de dibujado es importante, piensa qué estrella deberías dibujar en primer lugar. Nota: puedes definir dos vectores de índices, cada uno con un nombre distinto, y crear un *buffer* adicional en la función *initBuffers*.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

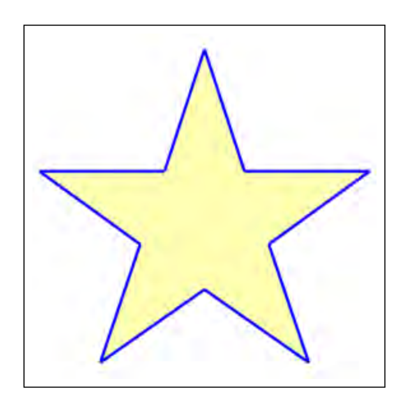


Figura 2.11: Ejemplo de dos estrellas: una aporta el color interior y la otra el color del borde

2.3.4. Variables *out*

Hasta el momento, cada vértice consta de un único atributo: su posición. Ahora se va a añadir un segundo atributo, por ejemplo, un valor de color para cada vértice. En primer lugar hay que ampliar la información de cada vértice para contener su valor de color. El listado 2.8 muestra el modelo de un triángulo tras añadir las cuatro componentes de color (RGBA) a cada vértice. También es necesario habilitar los dos atributos correspondientes, tal y como se muestra en el listado 2.9. Estúdialo y contesta a esta pregunta, ¿dónde colocarías este código en el listado 2.4?

Listado 2.8: Ejemplo de modelo con dos atributos por vértice: posición y color

```
var exampleTriangle = {

  "vertices" : [
      -0.7, -0.7, 0.0,     1.0, 0.0, 0.0, 1.0,   // rojo
       0.7, -0.7, 0.0,     0.0, 1.0, 0.0, 1.0,   // verde
       0.0,  0.7, 0.0,     0.0, 0.0, 1.0, 1.0],  // azul

  "indices" : [ 0, 1, 2]

};
```

Listado 2.9: Localización y habilitación de los dos atributos: posición y color

```
program.vertexPositionAttribute =
    gl.getAttribLocation(program, "VertexPosition");
gl.enableVertexAttribArray(program.vertexPositionAttribute);

program.vertexColorAttribute =
    gl.getAttribLocation(program, "VertexColor");
gl.enableVertexAttribArray(program.vertexColorAttribute);
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 2.10: Ejemplo de *shader* con dos atributos por vértice: posición y color

```
#version 300 es // Shader de vértices

in  vec3 VertexPosition;
in  vec4 VertexColor;
out vec4 colorOut;

void main()  {

  colorOut    = VertexColor;
  gl_Position = vec4(VertexPosition, 1.0);
}

#version 300 es // Shader de fragmentos
precision mediump float;

in  vec4 colorOut;
out vec4 fragmentColor;

void main() {

  fragmentColor = colorOut;
}
```

Los cambios correspondientes al *shader* se recogen en el listado 2.10. Observa que ahora están declarados en el *shader* de vértices los dos atributos de tipo `in`, `VertexPosition` y `VertexColor`. Por otra parte, la variable *colorOut*, en el *shader* de vértices, se ha declarado como variable de salida mediante la palabra clave `out`. A la variable *colorOut* simplemente se le asigna el valor de color por vértice representado con el atributo `VertexColor`. Las variables declaradas como variables de salida en el *shader* de vértices deben estar declaradas con el mismo nombre en el *shader* de fragmentos como variables de entrada (utilizando la palabra clave `in`) y serán interpoladas linealmente en el proceso de rasterización. También la variable `fragmentColor` es de tipo `out` y contiene el color de salida del *shader* de fragmentos. La figura 2.12 muestra el resultado.

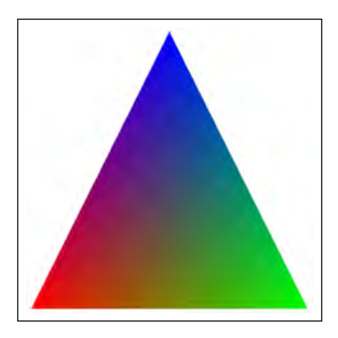


Figura 2.12: Resultado de visualizar un triángulo con un valor de color diferente para cada vértice

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Por último, hay que modificar la función de dibujo. Observa la nueva función en el listado 2.11 que recoge los cambios necesarios. Presta atención a los dos últimos parámetros de las dos llamadas a *gl.vertexAttribPointer*. Consulta la documentación para entender el por qué de esos valores.

Listado 2.11: Dibujo de un modelo con dos atributos por vértice

```
function draw(model) {

  gl.bindBuffer(gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer(program.vertexPositionAttribute, 3,
      gl.FLOAT, false, 7*4, 0);
  gl.vertexAttribPointer(program.vertexColorAttribute,    4,
      gl.FLOAT, false, 7*4, 3*4);

  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  gl.drawElements(gl.TRIANGLES, 3, gl.UNSIGNED_SHORT, 0);
}
```

Ejercicios

▶ **2.15** ¿Cuál es el número mínimo de *buffers* que necesitas en WebGL para almacenar un modelo poligonal que utiliza la representación de vértices compartidos si para cada vértice almacenas los atributos de posición y normal?

▶ **2.16** Observa la imagen de la figura 2.13, ¿qué colores se han asignado a los vértices? Edita *c02/interpola.js*, que ya incluye los fragmentos de código que se han explicado en esta sección, y modifica los valores de color para obtener ese mismo resultado. Carga en el navegador *c02/interpola.html* para comprobar si has acertado.

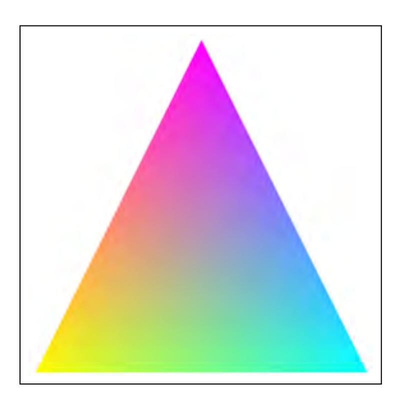


Figura 2.13: Observa el triángulo y contesta, ¿qué color se ha asignado a cada vértice?

▶ **2.17** Si modificas el programa *interpola* para que se muestre una estrella con color amarillo en las cinco puntas y color azul en el resto de vértices, ¿qué resultado crees que se obtendrá? Haz los cambios oportunos y comprueba si el resultado coincide con el esperado. Prueba también a cambiar los colores de los vértices de las puntas, y de los otros también, con los colores que tú elijas, experimenta, y obtén una combinación que sea de tu agrado.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

CUESTIONES

▶ **2.1** Algunos conceptos relacionados con los modelos poligonales son: teselado, triangularización, simplificación y optimización. Averigua el significado de dichos conceptos. Pon algún ejemplo donde resulte interesante aplicar cada uno de ellos.

▶ **2.2** ¿Por qué es importante hacer que todas las caras de una malla poligonal estén orientadas de la misma forma?

▶ **2.3** ¿Por qué es importante saber si un modelo poligonal es cerrado (que no presenta agujeros)?

▶ **2.4** ¿Qué es una tira de triángulos?

▶ **2.5** Supón que tienes dos tiras de triángulos donde cada tira representa un trozo diferente de la superficie de un modelo, y no hay vértices en común entre ambas tiras. La primera tira está formada por la siguiente secuencia de índices: 10, 11, 12, 13 y 14. Y la segunda tira por: 20, 21, 22 y 23. Si deseas unir ambas tiras para tener una única tira de triángulos, uniendo el final de la primera tira con el inicio de la segunda, determina la secuencia completa de índices: $10, 11, 12, 13, 14, ..., 20, 21, 22, 23$.

▶ **2.6** ¿Qué es un *buffer object*?

▶ **2.7** ¿Cuántos *buffer objects* necesitas como mínimo para almacenar un modelo poligonal? ¿Por qué?

▶ **2.8** ¿Qué significa el segundo parámetro de la orden `drawElements` si el primero es `gl.LINES`?

▶ **2.9** Para el modelo de un hexágono, ¿qué valor es el adecuado para el segundo parámetro de la orden `drawElements` para los casos en los que el primero sea `gl.LINES`, `gl.LINE_STRIP` y `gl.LINE_LOOP`?

▶ **2.10** ¿Qué significa que una variable sea `uniform`?

▶ **2.11** ¿Qué significa que una variable sea de tipo `in` en el *shader* de vértices? ¿Puede haber una variable de tipo `in` en el *shader* de fragmentos?

▶ **2.12** Si tienes una variable `uniform` para especificar el color de dibujo del modelo de una estrella, en el caso de querer dibujar varias estrellas pero todas ellas del mismo color, ¿es necesario llamar a `gl.uniform` antes de ordenar cada dibujado?

▶ **2.13** Si añades un nuevo atributo por vértice, ¿cómo especificas los valores para este nuevo atributo en el modelo?

▶ **2.14** ¿Qué es una variable de tipo `out` cuando esta aparece en el *shader* de vértices? ¿Y qué significa cuando aparece en el *shader* de fragmentos? ¿Crees que es posible que haya más de una variable de tipo `out` en el *shader* de vértices? ¿Y en el de fragmentos?

▶ **2.15** ¿Es posible establecer el valor de una variable de tipo `out` como, por ejemplo, se hace con las variables `uniform`?

▶ **2.16** Si tienes varios atributos por vértice, ¿cómo obtienes los dos últimos parámetros de la orden `vertexAttribPointer`?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 3

Transformaciones geométricas

Índice

3.1. Transformaciones básicas 56

3.1.1. Traslación 56

3.1.2. Escalado 56

3.1.3. Rotación 57

3.2. Concatenación de transformaciones 57

3.3. Matriz de transformación de la normal 58

3.4. Giro alrededor de un eje arbitrario 61

3.5. La biblioteca GLMATRIX 61

3.6. Transformaciones en WebGL 63

En la etapa de modelado los objetos se definen bajo un sistema de coordenadas propio. A la hora de crear una escena, estos objetos se incorporan bajo un nuevo sistema de coordenadas conocido como sistema de coordenadas del mundo. Este cambio de sistema de coordenadas es necesario y se realiza mediante transformaciones geométricas. La figura 3.1 muestra algunos ejemplos de objetos obtenidos mediante la aplicación de transformaciones a primitivas geométricas simples como el cubo, la esfera o el toro.

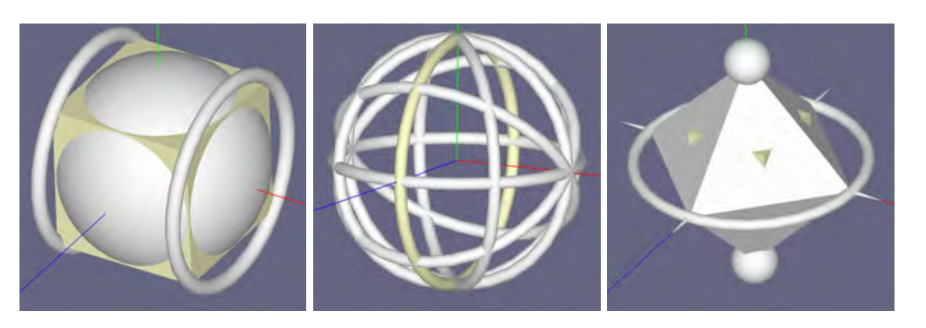


Figura 3.1: Ejemplos de objetos creados utilizando transformaciones geométricas

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

3.1. Transformaciones básicas

3.1.1. Traslación

La transformación de traslación consiste en desplazar el punto $p = (p_x, p_y, p_z)$ mediante un vector $t = (t_x, t_y, t_z)$, de manera que el nuevo punto $q = (q_x, q_y, q_z)$ se obtiene así:

$$q_x = p_x + t_x, \qquad q_y = p_y + t_y, \qquad q_z = p_z + t_z \tag{3.1}$$

La representación matricial con coordenadas homogéneas de esta transformación es:

$$T(t) = T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

Utilizando esta representación, el nuevo punto se obtiene así:

$$\tilde{q} = T(t) \cdot \tilde{p} \tag{3.3}$$

donde $\tilde{p} = (p_x, p_y, p_z, 1)^T$ y $\tilde{q} = (q_x, q_y, q_z, 1)^T$, es decir, los puntos p y q en coordenadas homogéneas.

3.1.2. Escalado

La transformación de escalado consiste en multiplicar el punto $p = (p_x, p_y, p_z)$ con los factores de escala s_x, s_y y s_z de tal manera que el nuevo punto $q = (q_x, q_y, q_z)$ se obtiene así:

$$q_x = p_x \cdot s_x, \qquad q_y = p_y \cdot s_y, \qquad q_z = p_z \cdot s_z \tag{3.4}$$

La representación matricial con coordenadas homogéneas de esta transformación es:

$$S(s) = S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.5}$$

Utilizando esta representación, el nuevo punto se obtiene así: $\tilde{q} = S(s) \cdot \tilde{p}$.

Ejercicios

▶ **3.1** Cuando los tres factores de escala son iguales, se denomina escalado uniforme. Ahora, lee y contesta las siguientes cuestiones:

- ¿Qué ocurre si los factores de escala son diferentes entre sí?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

- ¿Y si algún factor de escala es cero?
- ¿Qué ocurre si uno o varios factores de escala son negativos?
- ¿Y si el factor de escala está entre cero y uno?

3.1.3. Rotación

La transformación de rotación gira un punto un ángulo ϕ alrededor de un eje, y las representaciones matriciales con coordenadas homogéneas para los casos en los que el eje de giro coincida con uno de los ejes del sistema de coordenadas son las siguientes:

$$R_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

$$R_y(\phi) = \begin{pmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.7}$$

$$R_z(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

Utilizando cualquiera de estas representaciones, el nuevo punto siempre se obtiene así: $\tilde{q} = R(\phi) \cdot \tilde{p}$.

3.2. Concatenación de transformaciones

Una gran ventaja del uso de las transformaciones geométricas en su forma matricial con coordenadas homogéneas es que se pueden concatenar. De esta manera, una sola matriz puede representar toda una secuencia de matrices de transformación.

Cuando se realiza la concatenación de transformaciones, es muy importante operar la secuencia de transformaciones en el orden correcto, ya que el producto de matrices no posee la propiedad conmutativa.

Por ejemplo, piensa en una esfera con radio una unidad centrada en el origen de coordenadas y en las dos siguientes matrices de transformación T y S: $T(5, 0, 0)$ desplaza la componente x cinco unidades; $S(5, 5, 5)$ escala las tres componentes con un factor de cinco. Ahora, dibuja en el papel cómo quedaría la esfera después de aplicarle la matriz de transformación M si las matrices se multiplican de las dos

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

formas posibles, es decir, $M = T \cdot S$ y $M = S \cdot T$. Como verás, los resultados son bastante diferentes.

Por otra parte, el producto de matrices sí que posee la propiedad asociativa. Esto se puede aprovechar para reducir el número de operaciones, aumentando así la eficiencia.

3.3. Matriz de transformación de la normal

La matriz de transformación del modelo es consistente para geometría y para vectores tangentes a la superficie. Sin embargo, dicha matriz no siempre es válida para los vectores normales a la superficie. En concreto, esto ocurre cuando se utilizan transformaciones de escalado no uniforme (véase figura 3.2). En este caso, la matriz de transformación de la normal N es la traspuesta de la inversa de la matriz de transformación del modelo:

$$N = (M^{-1})^T \tag{3.9}$$

Además, como la normal es un vector y la traslación no le afecta (y el escalado y la rotación son transformaciones afines), solo hay que tomar los 3×3 componentes superior izquierda de M.

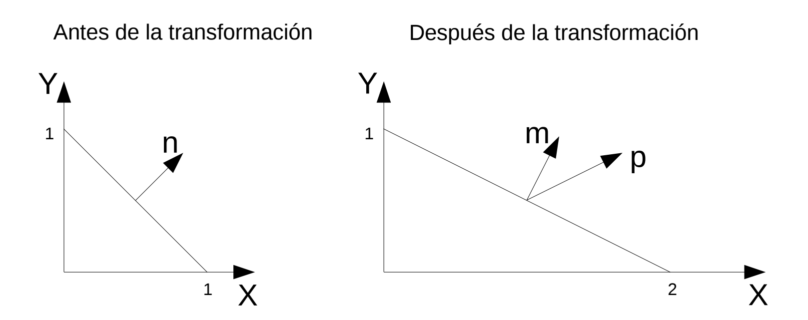


Figura 3.2: En la imagen de la izquierda se representa de perfil un polígono y su normal n. En la imagen de la derecha se muestra el mismo polígono tras aplicar una transformación de escalado no uniforme $S(2, 1)$. Si se aplica esta transformación a la normal n, se obtiene p como vector normal en lugar de m, que es la normal correcta

Señalar por último que, después de aplicar la transformación, y únicamente en el caso de incluir escalado, las longitudes de las normales no se preservan, por lo que es necesario normalizarlas. Como esta operación es cara computacionalmente, ¿se te ocurre alguna manera de evitarla?

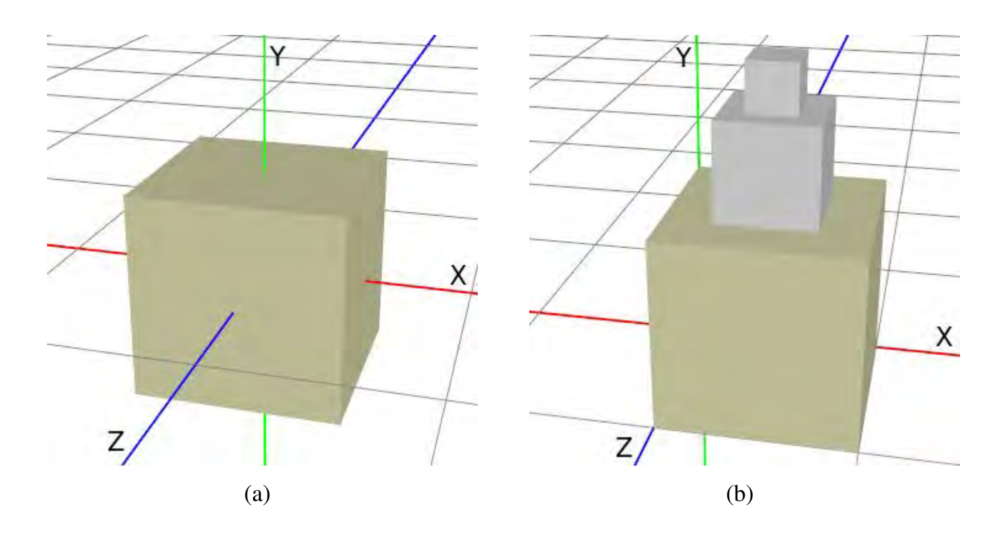
José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

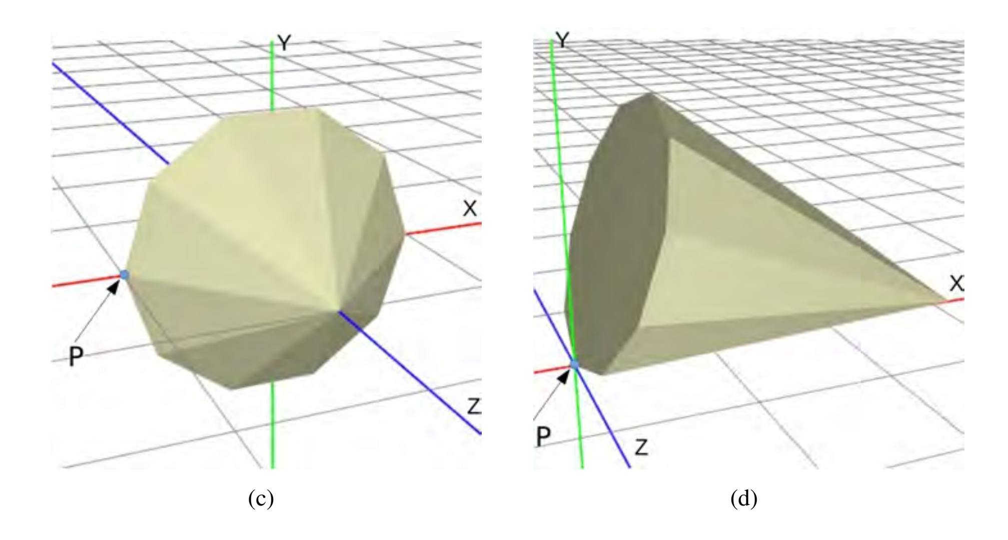
En los siguientes ejercicios el eje X es el de color rojo, el Y es el de color verde y el Z es el de color azul.

▶ **3.2** Comienza con un cubo de lado uno centrado en el origen de coordenadas, tal y como se muestra en la figura (*a*). Usa dos cubos más como este y obtén el modelo que se muestra en la figura (*b*), donde cada nuevo cubo tiene una longitud del lado la mitad de la del cubo anterior. Detalla las transformaciones utilizadas.



(a) (b)

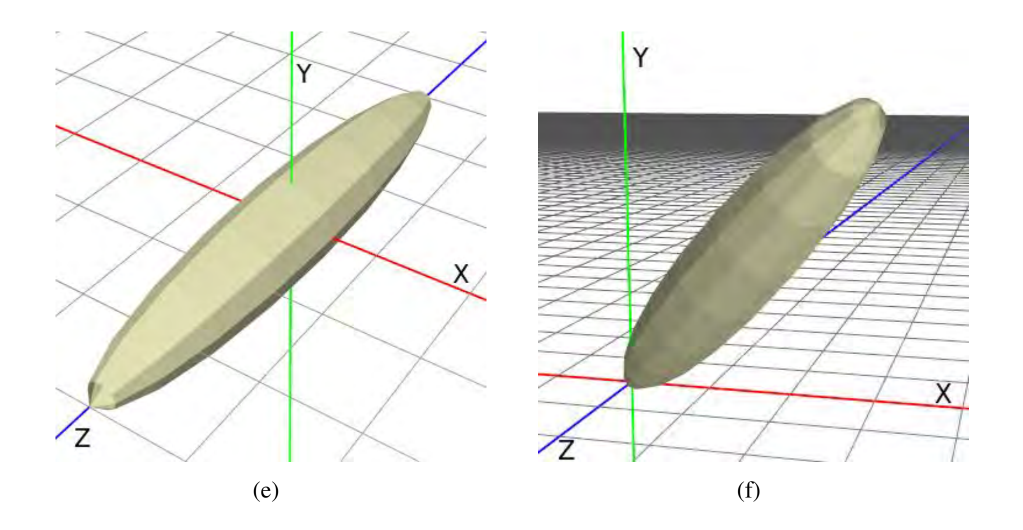
▶ **3.3** Determina las transformaciones que sitúan el cono que se muestra en la figura (*c*) (radio de la base y altura de valor 1) en la posición que se muestra en la figura (*d*) (radio de la base de valor 1 y altura de valor 3). Ten en cuenta el punto de referencia P señalado con una flecha, de manera que quede ubicado tal y como se observa en la figura.



(c) (d)

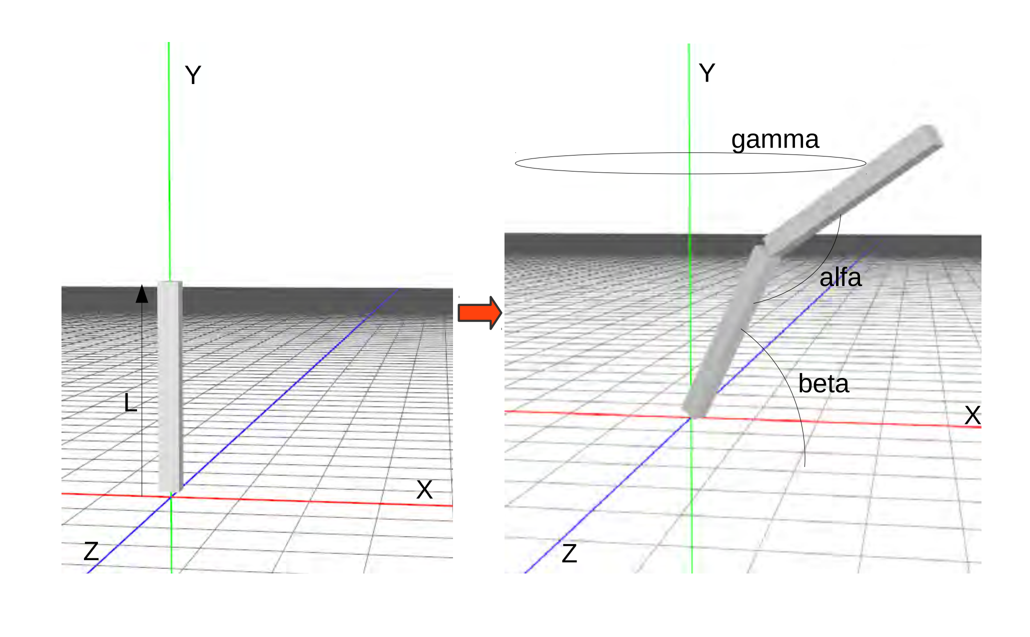
José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **3.4** Comienza con una esfera de radio uno centrada en el origen de coordenadas. La figura (e) muestra la esfera escalada con factores de escala $s_x = s_y = 0.5$ y $s_z = 3$. Obtén las transformaciones que situan a la esfera tal y como se muestra en la figura (f), donde un punto final está en el origen y el otro en la recta $x = y = z$.



(e) (f)

▶ **3.5** Observa la posición inicial del objeto en la figura de la izquierda. Este objeto lo vas a utilizar dos veces para crear el brazo articulado que se muestra en la figura de la derecha. Determina las transformaciones que has de aplicarle. La posición final del brazo ha de depender de los ángulos $alfa$, $beta$ y $gamma$.



José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

3.4. Giro alrededor de un eje arbitrario

Sean d y ϕ el vector unitario del eje de giro y el ángulo de giro, respectivamente. Para realizar la rotación hay que calcular en primer lugar una base ortogonal que contenga d. La idea es hacer un cambio de base entre la base que forman los ejes de coordenadas y la nueva base, haciendo coincidir el vector d con, por ejemplo, el eje X, para entonces rotar ϕ grados alrededor de X y finalmente deshacer el cambio de base.

La matriz que representa el cambio de base es esta:

$$R = \begin{pmatrix} d_x & d_y & d_z \\ e_x & e_y & e_z \\ f_x & f_y & f_z \end{pmatrix} \tag{3.10}$$

donde e es un vector unitario normal a d, y f es el producto vectorial de los otros dos vectores $f = d \times e$. Esta matriz deja al vector d en el eje X, al vector e en el eje Y y al vector f en el eje Z (véase figura 3.3). El vector e se puede obtener de la siguiente manera: partiendo del vector d, haz cero su componente de menor magnitud (el más pequeño en valor absoluto), intercambia los otros dos componentes, niega uno de ellos y normalízalo.

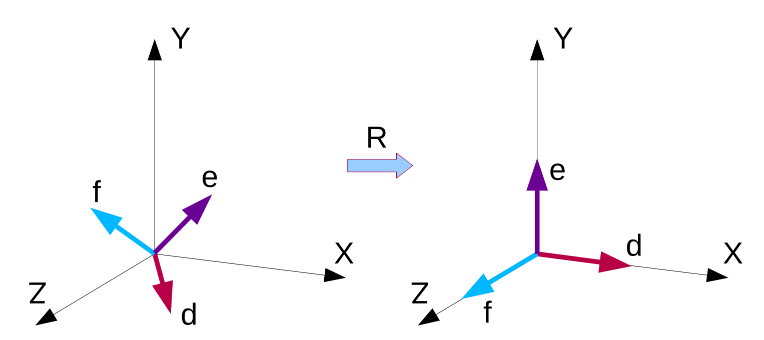


Figura 3.3: La nueva base formada por los vectores d, e y f se transforma para coincidir con los ejes de coordenadas

Así, teniendo en cuenta que R es ortogonal, es decir, que su inversa coincide con la traspuesta, la matriz de rotación final es:

$$R_d(\phi) = R^T R_x(\phi) R \tag{3.11}$$

3.5. La biblioteca GLMATRIX

Como ayuda a la programación, la biblioteca GLMATRIX proporciona funciones tanto para la construcción de las matrices de transformación como para operar con ellas. En concreto, las siguientes funciones permiten construir, respectivamente, las matrices de traslación, escalado y giro alrededor de un eje arbitrario que pasa por el origen:

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

- mat4.fromTranslation (out, v)
- mat4.fromScaling (out, v)
- mat4.fromRotation (out, rad, $axis$)

Ejercicios

▶ **3.6** Accede a la página web `glmatrix.net`. Ve a la documentación y, por ejemplo, busca el método `fromTranslation` de la clase `mat4`. La propia ayuda te permite acceder al código fuente de la librería. Ve al código del método `fromTranslation` y comprueba cómo construye la respectiva matriz de transformación.

Por ejemplo, la matriz de transformación M que escala un objeto al doble de su tamaño y después lo traslada en dirección del eje X un total de diez unidades, se obtendría de la siguiente forma:

```
var M = mat4.create();
var T = mat4.create();
var S = mat4.create();
mat4.fromTranslation (T, [10,0,0]);
mat4.fromScaling     (S, [2,2,2]);
mat4.multiply        (M, T, S);
```

Ejercicios

▶ **3.7** ¿Qué transformación produces si en el ejemplo anterior, en vez de `mat4.multiply (M, T, S)`, apareciese `mat4.multiply (M, S, T)`?

Si M es la matriz de transformación del modelo, para obtener la matriz de transformación de la normal, N, utilizando GLMATRIX se puede hacer, por ejemplo, lo siguiente:

```
var N = mat3.create();
mat3.fromMat4  (N,M);
mat3.invert    (N,N);
mat3.transpose(N,N);
```

GLMATRIX también porporciona la función `normalFromMat4(N, M)` que realiza exactamente las mismas operaciones que las del listado anterior.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

3.6. Transformaciones en WebGL

WebGL no proporciona modelos de primitivas geométricas 3D. Por lo tanto, y en primer lugar, es necesario obtener una descripción geométrica de primitivas básicas como el cubo, la esfera, el cono, etc. Por ejemplo, el listado 2.3 (del capítulo anterior) muestra el fragmento de código que define un cubo de lado 1 centrado en el origen de coordenadas, con sus caras paralelas a los planos de referencia XY, XZ e YZ. Observa que este modelo consta de dos vectores, uno contiene las coordenadas de los vértices y el otro contiene los índices al vector de vértices que de tres en tres describen los triángulos. El índice del primer vértice es 0. De momento se utilizan modelos que solo constan de geometría, es decir, no contienen otros atributos (normal, color, etc.), por lo que vamos a visualizarlos en alambre, es decir, solo las aristas (véase listado 3.1).

Listado 3.1: Visualización en alambre de un modelo poligonal definido con triángulos independientes

```
function draw(model) {

  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer (vertexPositionAttribute, 3,
                          gl.FLOAT, false, 0, 0);

  gl.bindBuffer (gl.ELEMENT_ARRAY_BUFFER, model.idBufferIndices);
  for (var i = 0; i < model.indices.length; i += 3)
    gl.drawElements (gl.LINE_LOOP, 3, gl.UNSIGNED_SHORT, i*2);

}
```

Observa ahora la nueva función `drawScene()` en el listado 3.2. Esta función incluye ahora los tres pasos necesarios para dibujar un modelo que requiere de una matriz de transformación:

1. Se obtiene la matriz de transformación del modelo, `modelMatrix`, que en este caso se trata de un escalado a la mitad de su tamaño.

2. Se establece el valor de la matriz de transformación en el *shader* de vértices.

3. Se ordena el dibujado del modelo. Será entonces cuando cada vértice del modelo se multiplique por la matriz de transformación especificada en el paso anterior.

El listado 3.2 también muestra comentadas las líneas que permiten obtener la matriz de la normal a partir de la matriz de transformación. Esta matriz también es enviada al *shader* de vértices para que sea operada con el atributo correspondiente.

En el *shader* de vértices se incluye la operación que multiplica cada vértice por su correspondiente matriz de transformación (véase listado 3.3). Observa que las líneas comentadas corresponden al código que haría falta en el caso de que, además, se suministrara el atributo de la normal para cada vértice.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 3.2: Ejemplo de los pasos necesarios para dibujar un objeto transformado

```
function drawScene() {

  gl.clear(gl.COLOR_BUFFER_BIT);

  // 1. calcula la matriz de transformación
  var modelMatrix  = mat4.create();
  mat4.fromScaling (modelMatrix, [0.5,0.5,0.5]);

  // 2. establece la matriz modelMatrix en el shader de vértices
  gl.uniformMatrix4fv(program.modelMatrixIndex, false,
      modelMatrix);

  // para la matriz de la normal:
  // var normalMatrix  = mat3.create();
  // mat3.normalFromMat4 (normalMatrix, modelMatrix);
  // gl.uniformMatrix3fv(program.normalMatrixIndex, false,
      normalMatrix);

  // 3. dibuja la primitiva
  draw(exampleCube);

}
```

Listado 3.3: *Shader* de vértices que opera cada vértice con la matriz de transformación del modelo

```
#version 300 es

uniform mat4 M;      // matriz de transformación del modelo
// uniform mat3 N; // matriz de transformación de la normal

in vec3 VertexPosition;
// in  vec3 VertexNormal;
// out vec3 VertexNormalT;

void main () {

  // VertexNormalT = normalize (N * VertexNormal);
  gl_Position = M * vec4(VertexPosition, 1.0);

}
```


ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

► **3.8** Edita *c03/transforma.html* y *c03/transforma.js*. Comprueba que se han incluido todos los cambios explicados en esta sección. Observa también *common/primitivasG.js*, que incluye los modelos de algunas primitivas geométricas simples. Echa un vistazo a la descripción de los modelos. Prueba a visualizar las diferentes primitivas.

► **3.9** Se dispone del modelo de un cubo definido con su centro en el origen de coordenadas, lado 1, y sus aristas paralelas a los ejes de coordenadas. Dado el siguiente fragmento de código en el que se especifican una serie de transformaciones geométricas, dibuja una vista frontal XY (no hace falta que dibujes la Z) de cómo quedan los dos cubos que se ordena dibujar teniendo en cuenta dichas transformaciones (asume que la funcion `drawCubo()` produce el dibujado del modelo del cubo).

```
var S  = mat4.create();
var T1 = mat4.create();
var T2 = mat4.create();
var M  = mat4.create();

mat4.fromScaling (S, [ 1, 2, 1]);
mat4.fromTranslation (T1, [ 0, 0.5, 0]);
ma4.multiply (M, S, T1);
gl.uniformMatrix4fv(program.modelMatrixIndex, false, M);
drawCubo();

mat4.fromTranslation (T2, [ 1, 0, 0]);
ma4.multiply (M, M, T2);
gl.uniformMatrix4fv(program.modelMatrixIndex, false, M);
drawCubo();
```

► **3.10** Observa la escena que se muestra en la figura 3.4. Crea una escena que produzca la misma salida utilizando únicamente la primitiva cubo. En primer lugar, piensa sobre el papel qué transformaciones necesitas aplicar y solo después procede con la escritura del código. El cubo central tiene de lado 0.1, y los que están a su lado tienen la misma base pero una altura que va incrementándose en 0.1 a medida que se alejan del central. Usa un bucle para pintar los trece cubos. **Nota:** ambas imágenes muestran la misma escena, pero en la imagen de la derecha se ha girado la escena para apreciar mejor que son objetos 3D.

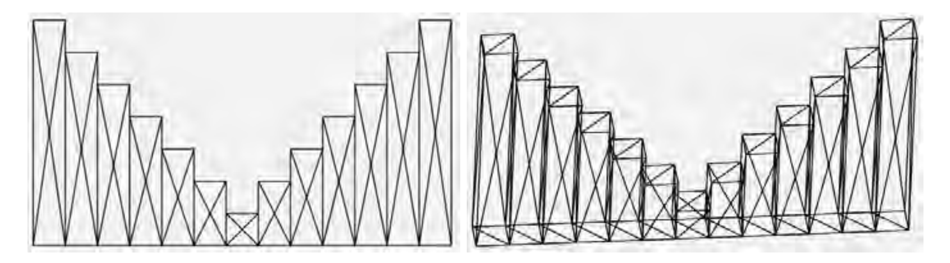


Figura 3.4: Escena modelada a partir de la primitiva geométrica cubo

► **3.11** Modela la típica grúa de obra cuyo esquema se muestra en la figura 3.5 utilizando como única primitiva cubos de lado 1 centrados en el origen de coordenadas. La carga es de lado 1, el contrapeso es de lado 1.4, y tanto el pie como el brazo tienen 10 de longitud.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Incluye las transformaciones que giran la grúa sobre su pie, que desplazan la carga a lo largo del brazo, y que levantan y descienden la carga.

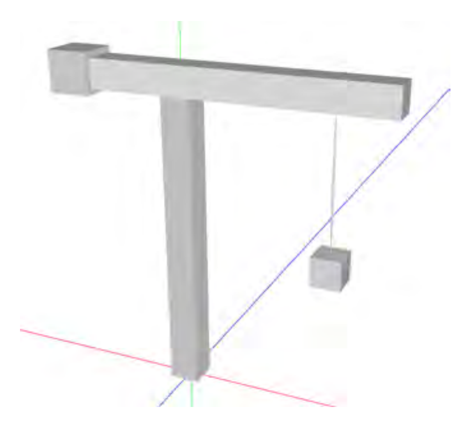


Figura 3.5: Ejemplo de una grúa de obra modelada con cubos y transformaciones geométricas

▶ **3.12** En este ejercicio vas a hacer un tanque. En la figura 3.6 puedes observar lo que has de conseguir. Los cilindros de las ruedas tienen una altura de 0.8 y un radio de 0.1. Tapa los cilindros por ambos lados. El cuerpo del tanque es un cubo escalado para ocupar lo mismo que las ruedas, y una altura de 0.2. La torreta del cañón es un cilindro de radio 0.2 y altura 0.8. Observa dónde está situada y haz coincidir la tuya en la misma posición. Tápala también. Por último, añade el cañón, que es un cilindro de longitud 0.8 y radio 0.03.

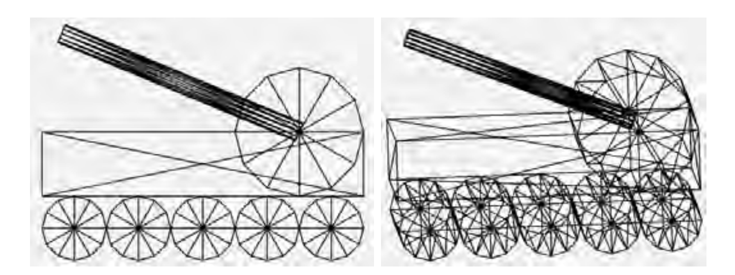


Figura 3.6: Tanque modelado a partir de diferentes primitivas geométricas

▶ **3.13** Crea un reloj como el de la figura 3.7. No te doy dimensiones, elígelas tú. Imítalo en la medida de lo posible. Sí que te pido que marque las dos en punto.

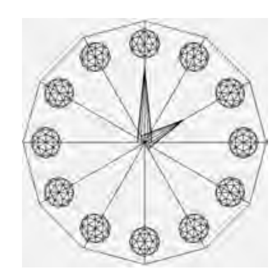


Figura 3.7: Reloj modelado a partir de primitivas geométricas básicas

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **3.14** Indica la secuencia de transformaciones que debes aplicar a un cubo centrado en el origen de coordenadas (y sus lados paralelos a los ejes de coordenadas) para que gire 30 grados alrededor de su diagonal (del (-1,-1,-1) al (1,1,1)). Solo puedes usar giros alrededor de los ejes de coordenadas.

▶ **3.15** Internet es una fuente de inspiración excepcional, utilízala para buscar ejemplos que te sirvan de muestra y practicar con el uso de las transformaciones geométricas. Fíjate en los que aparecen en la figura 3.8.



Figura 3.8: Otros ejemplos de juegos de bloques de madera coloreados

---

CUESTIONES

▶ **3.1** ¿Qué es la matriz de transformación del modelo?

▶ **3.2** ¿Dónde operas la matriz de transformación del modelo con cada uno de sus vértices?

▶ **3.3** ¿Por qué en el *shader* se declara la matriz de transformación del modelo de tipo *uniform*?

▶ **3.4** ¿En qué tipo de *shader*, vértices o fragmentos, utilizas la matriz de transformación del modelo?

▶ **3.5** ¿Qué es la matriz de transformación de la normal? ¿Dónde operas cada normal con su matriz de transformación?

▶ **3.6** ¿Por qué las transformaciones de traslación se pueden omitir en el proceso de cálculo de la matriz de transformación de la normal?

▶ **3.7** Dada una matriz de transformación que al operarla con un modelo produce su simétrico, ¿qué le ocurren a las normales si las operamos con esa misma matriz? ¿produce las normales correctas?

▶ **3.8** ¿Qué hace exactamente la función `create` de la librería GLMATRIX?

▶ **3.9** Si tienes tres transformaciones representadas cada una de ellas por las matrices mA, mB y mC, ¿cómo debes operar las matrices para obtener la matriz de transformación del modelo cuando la primera transformación que deseas aplicar es mB, después mA y por último mC?

▶ **3.10** Dado un cilindro cuya base está sobre el plano Z = 0 y su eje en el +Z, radio uno y altura uno, obtén la matriz de transformación, T, que aplicada sobre dicho cilindro lo lleva a que su eje coincida con la diagonal positiva $X = Y, Z = 0$ teniendo en cuenta que el punto centro de la base que residía en el origen de coordenadas permanece en la misma posición después de su transformación.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **3.11** Imagina que necesitas visualizar un modelo poligonal creado por un tercero. Del modelo tienes su descripción geométrica (vértices, normales, índices, etc.). Por desgracia, al tratar de visualizarlo con WebGL sin aplicarle ningún tipo de transformación geométrica obtienes un canvas vacío. Indica qué transformaciones aplicarías sobre el modelo a fin de conseguir visualizarlo de manera que este aparezca completo dentro del canvas y al mismo tiempo se dibuje lo más grande posible. Indica también cómo obtendrías los valores que necesitas para realizar dichas transformaciones.

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 4

Viendo en 3D

Índice

4.1. Transformación de la cámara **69**
4.2. Transformación de proyección **71**
4.2.1. Proyección paralela 71
4.2.2. Proyección perspectiva 73
4.3. Transformación al área de dibujo **74**
4.4. Eliminación de partes ocultas **74**
4.5. Viendo en 3D con WebGL **75**

Al igual que en el mundo real se utiliza una cámara para conseguir fotografías, en nuestro mundo virtual también es necesario definir un modelo de cámara que permita obtener vistas 2D de nuestro mundo 3D. El proceso por el que la cámara sintética obtiene una fotografía se implementa como una secuencia de tres transformaciones:

- Transformación de la cámara: ubica la cámara virtual en el origen del sistema de coordenadas orientada de manera conveniente.
- Transformación de proyección: determina cuánto del mundo 3D es visible. A este espacio limitado se le denomina *volumen de la vista* y transforma el contenido de este volumen al volumen canónico de la vista.
- Transformación al área de dibujo: el contenido del volumen canónico de la vista se transforma para ubicarlo en el espacio de la ventana destinado a mostrar el resultado de la vista 2D.

4.1. Transformación de la cámara

La posición de una cámara (el lugar desde el que se va a tomar la fotografía), se establece especificando un punto p del espacio 3D. Una vez posicionada, la cámara se orienta de manera que su objetivo quede apuntando a un punto específico

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

de la escena. A este punto i se le conoce con el nombre de *punto de interés*. En general, los fotográfos utilizan la cámara para hacer fotos apaisadas u orientadas en vertical, aunque tampoco resulta extraño ver fotografías tomadas con otras inclinaciones. Esta inclinación se establece mediante el vector UP denominado *vector de inclinación*. Con estos tres datos queda perfectamente posicionada y orientada la cámara, tal y como se muestra en la figura 4.1.

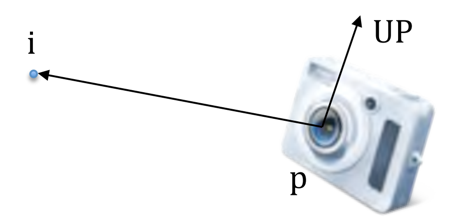


Figura 4.1: Parámetros para ubicar y orientar una cámara: p, posición de la cámara; UP, vector de inclinación; i, punto de interés

Algunas de las operaciones que los sistemas gráficos realizan requieren que la cámara esté situada en el origen de coordenadas, apuntando en la dirección del eje Z negativo y coincidiendo el vector de inclinación con el eje Y positivo. Por esta razón, es necesario aplicar una transformación al mundo 3D de manera que, desde la posición y orientación requeridas por el sistema gráfico, se observe lo mismo que desde donde el usuario estableció su cámara (véase figura 4.2). A esta transformación se le denomina *transformación de la cámara*.

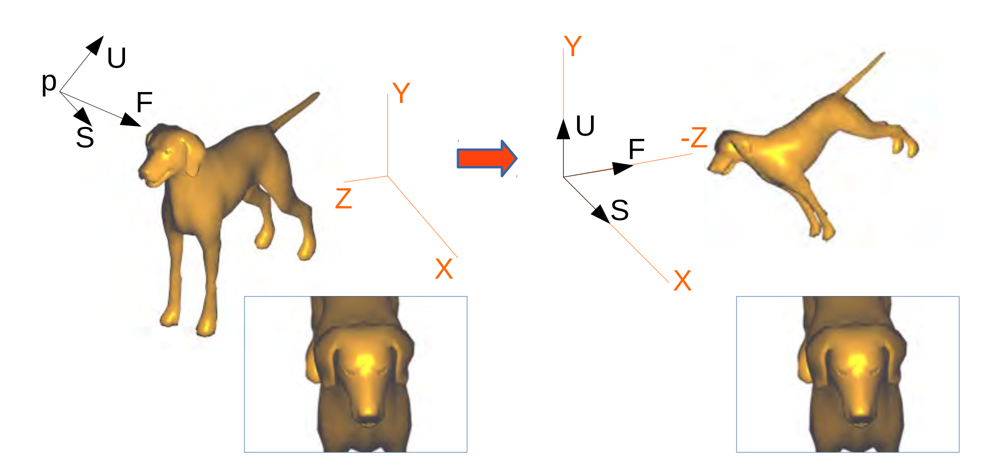


Figura 4.2: Transformación de la cámara. La cámara situada en el punto p en la imagen de la izquierda se transforma para quedar como se observa en la imagen de la derecha. Dicha transformación se aplica al objeto de tal manera que lo que se observa sea lo mismo en ambas situaciones

Si F es el vector normalizado que desde la posición de la cámara apunta al punto de interés, UP' es el vector de inclinación normalizado, $S = F \times UP'$ y $U = S \times F$, entonces el resultado de la siguiente operación crea la matriz de

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

transformación M_C que sitúa la cámara en la posición y orientación requeridas por el sistema gráfico:

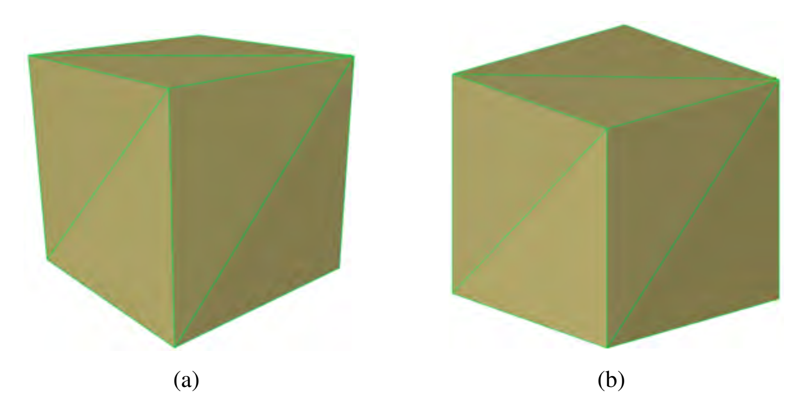
$$M_C = \begin{pmatrix} S_x & S_y & S_z & 0 \\ U_x & U_y & U_z & 0 \\ -F_x & -F_y & -F_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

4.2. Transformación de proyección

El volumen de la vista determina la parte del mundo 3D que puede ser vista por el observador. La forma y dimensión de este volumen depende del tipo de proyección:

- Proyección perspectiva. Es similar a como funciona nuestra vista y se utiliza para generar imágenes más fieles a la realidad en aplicaciones como videojuegos, simulaciones o, en general, la mayor parte de aplicaciones gráficas.
- Proyección paralela. Es la utilizada principalmente en ingeniería o arquitectura. Se caracteriza por preservar longitudes y ángulos.

La figura 4.3 muestra un ejemplo de un cubo dibujado con ambos tipos de vistas.



(a) (b)

Figura 4.3: Vista de un cubo obtenida con: (a) proyección perspectiva y (b) proyección paralela

4.2.1. Proyección paralela

Este tipo de proyección se caracteriza por que los rayos de proyección son paralelos entre sí e intersectan de forma perpendicular con el plano de proyección. El volumen de la vista tiene forma de caja, la cual, se alinea con los ejes de coordenadas tal y como se muestra en la figura 4.4, donde también se han indicado los nombres de los seis parámetros que definen dicho volumen.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

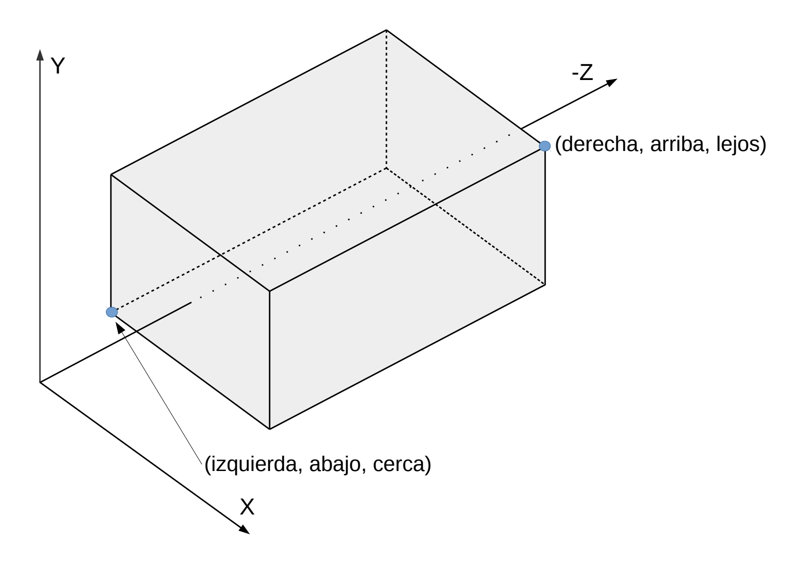


Figura 4.4: Esquema del volumen de la vista de una proyección paralela

En general, los sistemas gráficos trasladan y escalan esa caja de manera que la convierten en un cubo centrado en el origen de coordenadas. A este cubo se le denomina *volumen canónico de la vista* (véase figura 4.5) y a las coordenadas en este volumen *coordenadas normalizadas del dispositivo*. La matriz de transformación correspondiente para un cubo de lado 2 es la siguiente:

$$M_{par} = \begin{pmatrix} \frac{2}{derecha-izquierda} & 0 & 0 & -\frac{derecha+izquierda}{derecha-izquierda} \\ 0 & \frac{2}{arriba-abajo} & 0 & -\frac{arriba+abajo}{arriba-abajo} \\ 0 & 0 & \frac{2}{lejos-cerca} & -\frac{lejos+cerca}{lejos-cerca} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

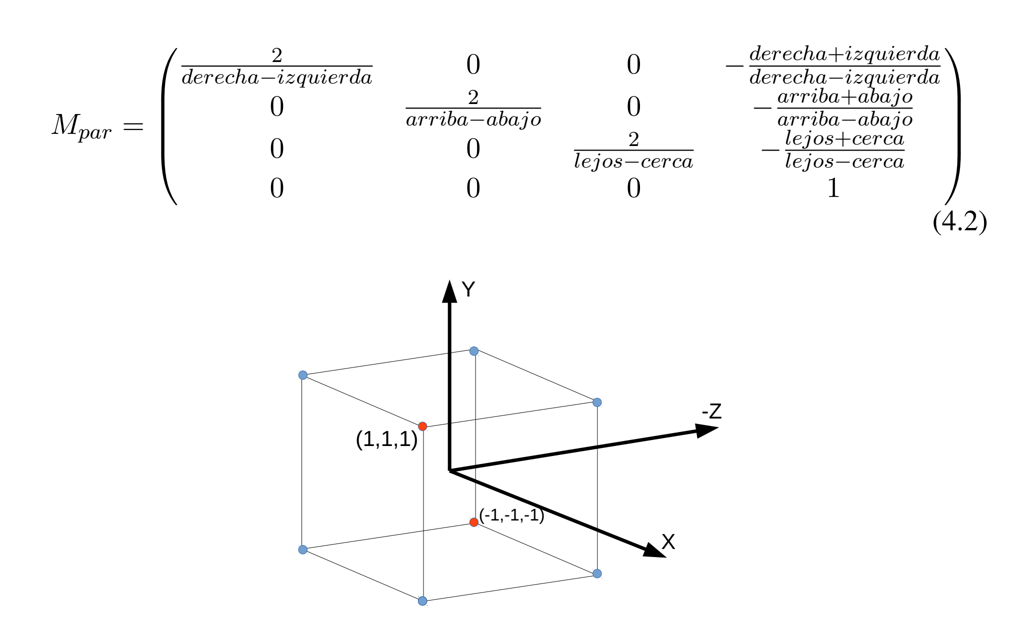


Figura 4.5: Volumen canónico de la vista, cubo de lado 2 centrado en el origen de coordenadas

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

4.2.2. Proyección perspectiva

Este tipo de proyección se caracteriza por que los rayos de proyección parten todos ellos desde la posición del observador. El volumen de la vista tiene forma de pirámide truncada, que queda definida mediante cuatro parámetros: los planos cerca y lejos (los mismos que en la proyección paralela), el ángulo θ en la dirección Y y la relación de aspecto de la base de la pirámide $ancho/alto$. En la figura 4.6 se detallan estos parámetros.

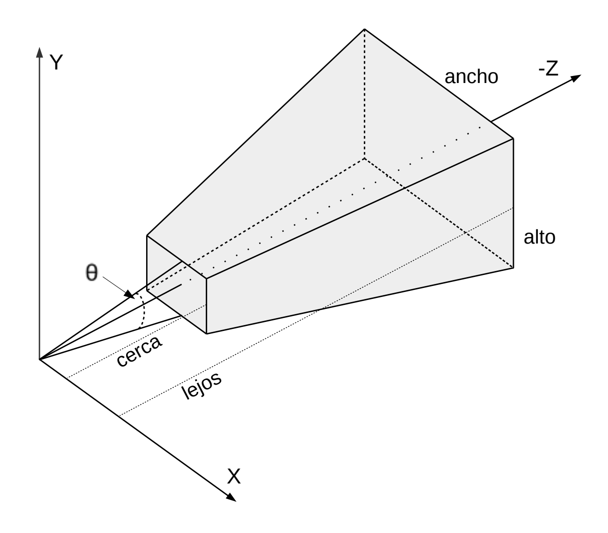


Figura 4.6: Esquema del volumen de la vista de una proyección perspectiva

En general, los sistemas gráficos convierten ese volumen con forma de pirámide en el volumen canónico de la vista. La matriz de transformación correspondiente para un cubo de lado 2 es la siguiente:

$$M_{per} = \begin{pmatrix} \frac{1}{aspect \cdot tan(\theta/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\theta/2)} & 0 & 0 \\ 0 & 0 & \frac{lejos+cerca}{cerca-lejos} & \frac{2 \cdot lejos \cdot cerca}{cerca-lejos} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad (4.3)$$

Ejercicios

► **4.1** El resultado de una proyección perspectiva depende, entre otras cosas, de la distancia de la cámara a la escena. Es decir, el resultado de la proyección cambia si la cámara se acerca o se aleja de la escena. Sin embargo, si se utiliza proyección paralela su resultado es independiente de dicha distancia. Si deseas utilizar proyección paralela y al mismo tiempo simular el efecto de que, por ejemplo, al acercar la cámara a la escena la proyección aumente, ¿cómo lo harías?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

4.3. Transformación al área de dibujo

El área de dibujo, también conocida por su término en inglés *viewport*, es la parte de la ventana de la aplicación donde se muestra la vista 2D. Por ejemplo, la figura 4.7 muestra un canvas con tres áreas de dibujo donde para cada una de ellas se muestra la misma escena pero vista desde una posición y orientación diferentes. La transformación al *viewport* consiste en mover el resultado de la proyección a dicha área. Se asume que la geometría a visualizar reside en el volumen canónico de la vista, es decir, se cumple que las coordenadas de todos los puntos $(x, y, z) \in [-1, 1]^3$. Entonces, si n_x y n_y son respectivamente el ancho y el alto del área de dibujo en píxeles, y o_x y o_y son las coordenadas de ventana del píxel de la esquina inferior izquierda del área de dibujo, para cualquier punto que resida en el volumen canónico de la vista, sus coordenadas de ventana se obtienen con la siguiente transformación:

$$M_{vp} = \begin{pmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} + o_x \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} + o_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.4}$$

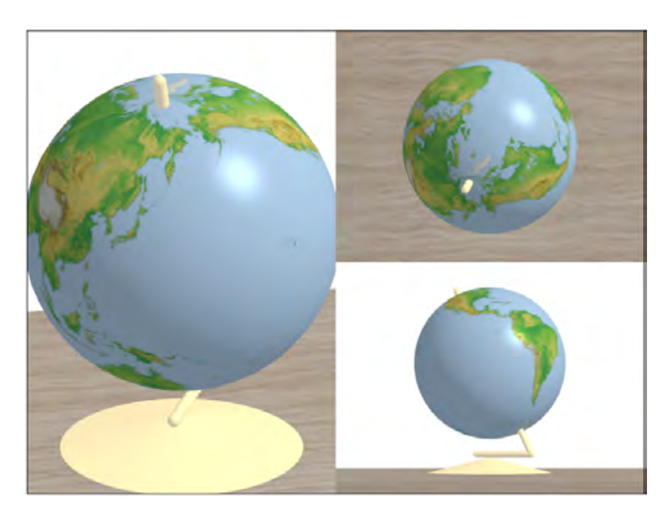


Figura 4.7: Ejemplo de canvas con tres áreas de dibujo o *viewports*. La escena es la misma en las tres áreas, sin embargo, la transformación de la cámara es distinta para cada área.

4.4. Eliminación de partes ocultas

La eliminación de partes ocultas consiste en determinar qué primitivas de la escena son tapadas por otras primitivas desde el punto de vista del observador (véase figura 4.8). Aunque para resolver este problema se han desarrollado diversos algoritmos, el más utilizado en la práctica es el algoritmo conocido como *z-buffer*.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

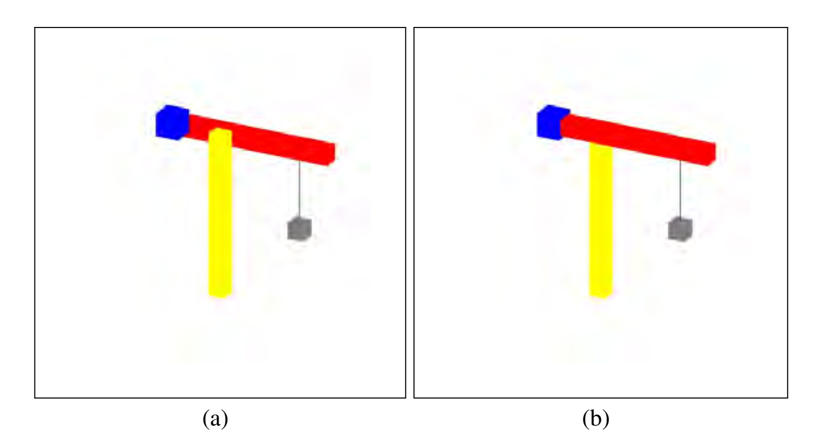


Figura 4.8: Ejemplo de escena visualizada: (a) sin resolver el problema de la visibilidad y (b) con el problema de la visibilidad resuelto

El algoritmo *z-buffer* se caracteriza por su simplicidad. Para cada píxel de la primitiva que se está dibujando, su valor de profundidad (su coordenada z) se compara con el valor almacenado en un *buffer* denominado *buffer de profundidad*. Si la profundidad de la primitiva para dicho píxel es menor que el valor almacenado en el *buffer* para ese mismo píxel, tanto el *buffer* de color como el de profundidad se actualizan con los valores de la nueva primitiva, siendo eliminado en cualquier otro caso.

El algoritmo se muestra en el listado 4.1. En este algoritmo no importa el orden en que se pinten las primitivas, pero sí es muy importante que el *buffer* de profundidad se inicialice siempre al valor de profundidad máxima antes de pintar la primera primitiva.

Listado 4.1: Algoritmo del *z-buffer*

```
if (pixel.z < bufferProfundidad(x,y).z) {

  bufferProfundidad(x,y).z = pixel.z;
  bufferColor(x,y).color    = pixel.color;

}
```

4.5. Viendo en 3D con WebGL

Hasta ahora, en los ejercicios realizados en los capítulos anteriores, se ha conseguido visualizar modelos sin tener que realizar ni la transformación de la cámara ni establecer un tipo de proyección. Esto ocurre porque la escena a visualizar se ha definido de manera que quedase dentro del volumen canónico de la vista. De esta

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

forma se obtiene una proyección paralela del contenido del volumen observando la escena en dirección del eje $-Z$.

En este capítulo se ha visto cómo construir la matriz de transformación de la cámara para poder observar la escena desde cualquier punto de vista, y la matriz de proyección para poder elegir entre vista paralela y vista perspectiva. En WebGL es responsabilidad del programador calcular estas matrices y operarlas con cada uno de los vértices del modelo.

Habitualmente, la matriz de la cámara se opera con la de transformación del modelo, creando la transformación conocida con el nombre de *modelo-vista*. Esta matriz y la de proyección se suministran al procesador de vértices, donde cada vértice v debe ser multiplicado por ambas matrices. Si M_C es la matriz de transformación de la cámara, M_M es la matriz de transformación del modelo, y M_{MV} es la matriz modelo-vista, es decir $M_{MV} = M_C \cdot M_M$, el nuevo vértice v' se obtiene como resultado de la siguiente operación: $v' = M_{Proy} \cdot M_{MV} \cdot v$; donde M_{Proy} será M_{Par} o M_{Per}. El listado 4.2 recoge estas operaciones.

Listado 4.2: *Shader* de vértices para transformar la posición de cada vértice

```
#version 300 es

uniform  mat4 projectionMatrix;  // perspectiva o paralela
uniform  mat4 modelViewMatrix;   // cameraMatrix * modelMatrix

in vec3 VertexPosition;

void main()  {

  gl_Position = projectionMatrix *
                modelViewMatrix *
                vec4(VertexPosition ,1.0);

}
```

La librería GLMATRIX proporciona diversas funciones para construir las matrices vistas en este capítulo. Así, la función *mat4.lookAt* construye la matriz de transformación de la cámara (ecuación 4.1), a partir de la posición de la cámara p, el punto de interés i y el vector de inclinación UP. Además, las funciones *mat4.ortho* y *mat4.perspective* construyen las matrices que transforman el volumen de la vista de una proyección paralela y perspectiva, respectivamente, al volumen canónico de la vista. Son estas:

- `mat4.lookAt (out, p, i, UP)`
- `mat4.ortho (out, izquierda, derecha, abajo, arriba, cerca, lejos)`
- `mat4.perspective (out,` θ`, ancho/alto, cerca, lejos)`

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Tras el procesamiento de los vértices, estos se reagrupan dependiendo del tipo de primitiva que se esté dibujando. Esto ocurre en la etapa de procesado de la primitiva (véase figura 4.9). A continuación se realiza la operación conocida con el nombre de *división perspectiva*, que consiste en que cada vértice sea dividido por su propia w (la cuarta coordenada del vértice). Esto es necesario, ya que el resultado de la proyección perspectiva puede hacer que la coordenada w del vértice sea distinta de 1.

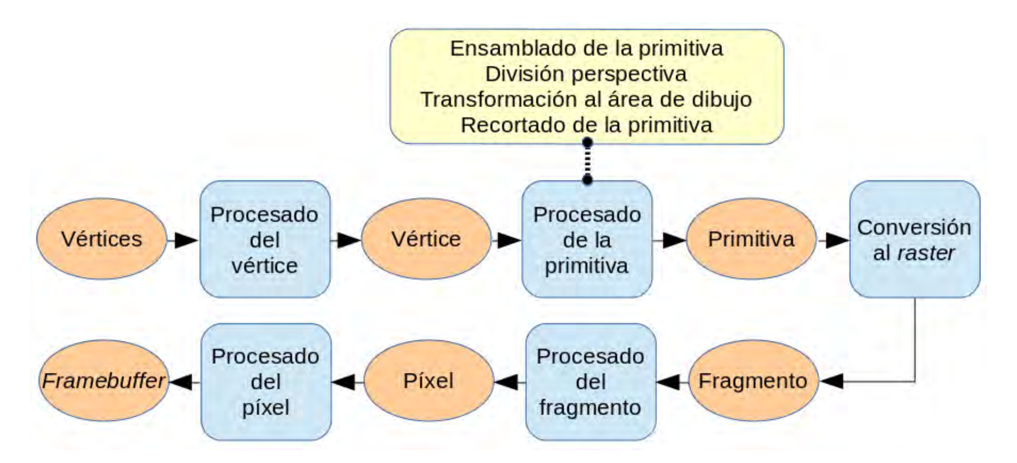


Figura 4.9: En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado de la primitiva

La transformación al área de dibujo se realiza a continuación, aún en la misma etapa. El programador solo debe especificar la ubicación del *viewport* en el canvas mediante la orden *gl.viewport*, indicando la esquina inferior izquierda, el ancho y el alto en coordenadas de ventana:

- `gl.viewport` (x, y, *ancho*, *alto*)

La figura 4.10 muestra el origen y tamaño de cada una de las tres áreas de dibujo del ejemplo de la figura 4.7 teniendo en cuenta que el canvas es de 800×600 píxeles.

Además, la llamada a la función *gl.viewport* debe realizarse cada vez que se produzca un cambio en el tamaño del canvas con el fin de mantener el tamaño del *viewport* actualizado. Es muy importante que la relación de aspecto del *viewport* sea igual a la relación de aspecto utilizada al definir el volumen de la vista para no deformar el resultado de la proyección (véase figura 4.11).

La última tarea dentro de la etapa de procesado de la primitiva es la operación de recortado, que consiste en eliminar los elementos que quedan fuera de los límites establecidos por el volumen de la vista. Si una primitiva intersecta con el volumen de la vista, se recorta de manera que por el *pipeline* únicamente continúen los trozos que han quedado dentro del volumen de la vista. Por ejemplo, la figura 4.12 muestra un cono en el que su vértice queda fuera del volumen de la vista produciendo el efecto de cono truncado.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

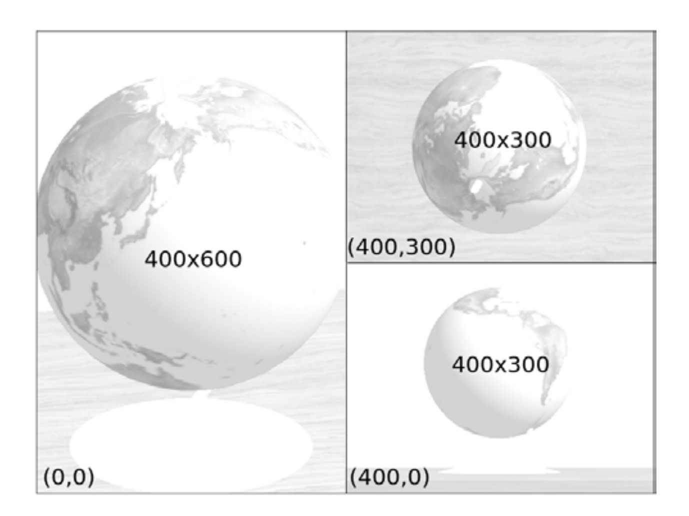


Figura 4.10: Ejemplo de tres áreas de dibujo o *viewports* en los que se especifica para cada uno de ellos su origen y tamaño

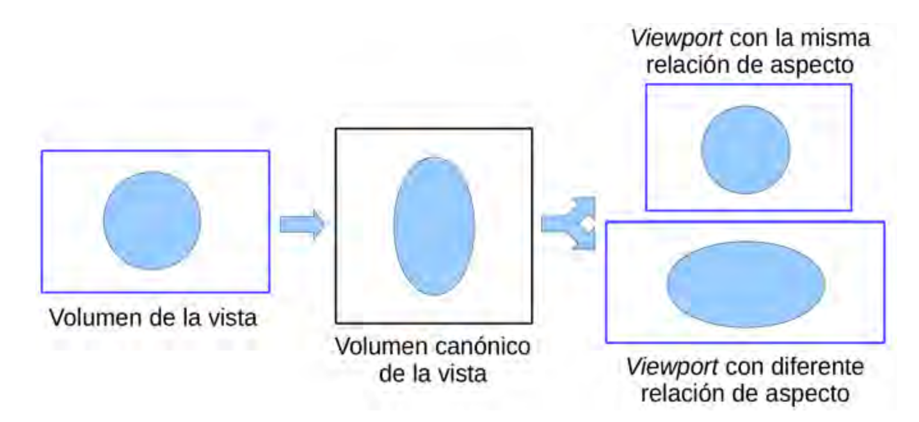


Figura 4.11: Si la proporción del volumen de la vista y del área de dibujo (*viewport*) coinciden no se producirá ningún tipo de deformación en la escena visualizada

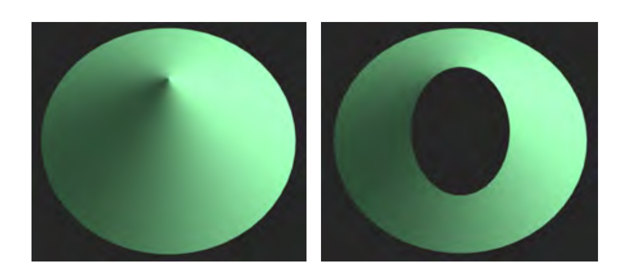


Figura 4.12: El vértice del cono en la imagen de la derecha queda fuera del volumen de la vista y se puede apreciar el resultado de la operación de recortado

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Respecto a la eliminación de partes ocultas, WebGL implementa el algoritmo del *z-buffer* y la GPU comprueba la profundidad de cada fragmento de forma fija en la etapa de procesado del fragmento, pero después de ejecutar el *shader* de fragmentos (véase figura 4.13). A esta comprobación se le denomina test de profundidad. Aunque esta operación está implementada en la GPU, el programador aún debe realizar dos tareas:

1. Habilitar la operación del test de profundidad (ya que por defecto no se realiza):
   - `gl.enable (gl.DEPTH_TEST);`
2. Inicializar el *buffer* a la profundidad máxima antes de comenzar a dibujar:
   - `gl.clear (... | gl.DEPTH_BUFFER_BIT);`

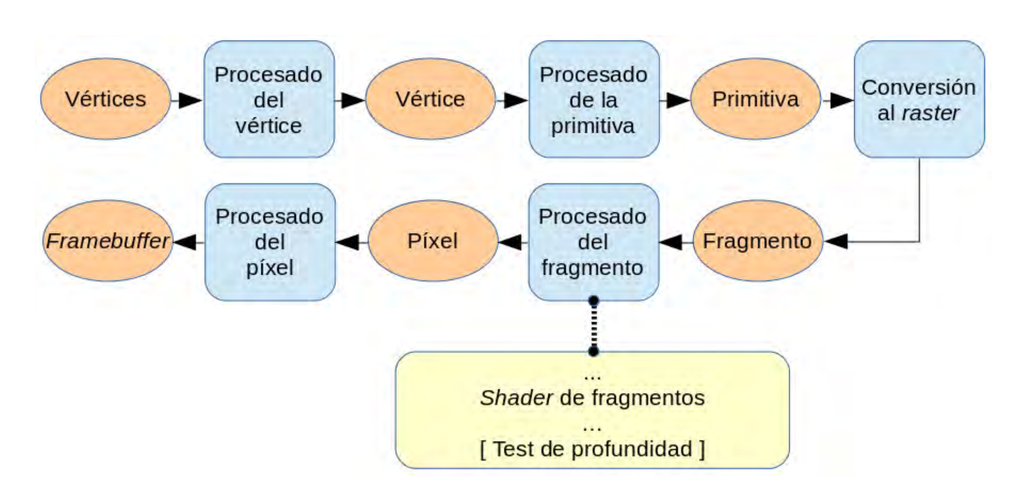


Figura 4.13: En color amarillo, las tareas principales que se realizan en la etapa correspondiente al procesado del fragmento donde se indica que el test de profundidad se realiza con posterioridad a la ejecución del *shader* de fragmentos y de manera opcional

Ejercicios

▶ **4.2** Carga la página *c04/mueveLaCamara.html*. Comprueba que se ha añadido una cámara interactiva que puedes modificar utilizando el ratón y la tecla *shift*. Edita el fichero *c04/mueveLaCamara.js* y estudia la función *getCameraMatrix*, que devuelve la matriz de transformación de la cámara, ¿cuál es el punto de interés establecido? Estudia también la función *setProjection*, ¿qué tipo de proyección se utiliza?, ¿qué cambios harías para utilizar el otro tipo de proyección visto en este capítulo?

▶ **4.3** Amplía la solución del ejercicio anterior para que se pueda cambiar de proyección paralela a perspectiva, y viceversa, y que sin modificar la matriz de transformación de la cámara se siga observando el modelo completo.

▶ **4.4** Parte, por ejemplo, del modelo del tanque que ya hiciste en el capítulo anterior y modifícalo para que en el canvas se muestren cuatro áreas de dibujo. Utiliza proyección

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

paralela en las cuatro áreas pero haz que en tres de ellas la dirección de la vista coincida con uno de los ejes de coordenadas y que la cuarta sea en dirección (1,1,1). En todas ellas el modelo debe observarse en su totalidad.

▶ **4.5** Amplía la solución del ejercicio 4.4 para que se realice la eliminación de las partes ocultas. Para observarlo mejor, utiliza un color diferente para cada primitiva y dibújalas como triángulos, no como líneas.

CUESTIONES

▶ **4.1** ¿Qué es la matriz de transformación de proyección? ¿y de qué tipos conoces?

▶ **4.2** ¿Qué significa cada uno de los parámetros de la función `perspective`? Dibújalo en papel.

▶ **4.3** ¿Qué función de la librería *glMatrix* utilizarás si quieres trabajar con proyección paralela?

▶ **4.4** ¿Qué función de la librería *glMatrix* te permite obtener fácilmente la matriz de transformación de la cámara?

▶ **4.5** ¿Qué dos matrices se operan habitualmente entre ellas para crear la matriz modelo-vista?

▶ **4.6** En el *shader* de vértices, ¿en qué orden debes operar cada vértice con las matrices de proyección, cámara y modelo?

▶ **4.7** Si el usuario modifica de manera interactiva la posición de la cámara, está claro que para obtener la nueva vista debes obtener la nueva matriz de transformación de la cámara pero, ¿te sigue valiendo la matriz de transformación de proyección que ya tenías o por el contrario también debes obtenerla?

▶ **4.8** Con el test de profundidad habilitado, describe qué ocurre si al dibujar no inicializas a la máxima profundidad el *buffer* correspondiente.

▶ **4.9** Ordena de manera temporal (primero la que se hace antes) las siguientes tareas que se realizan en el *pipeline* del sistema gráfico: transformación al área de dibujo, test de profundidad, transformación de la cámara, división perspectiva y asignación de color al fragmento.

▶ **4.10** Explica el problema que aparece cuando la razón de aspecto definida en el modelo de cámara es diferente de la relación de aspecto del área de dibujo. Pon algún ejemplo que lo ilustre. Explica qué transformaciones están involucradas en ese proceso. Explica también cómo resolverías este problema.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 5

Modelos de iluminación y sombreado

Índice

5.1. Modelo de iluminación de Phong 82
5.1.1. Luz ambiente 83
5.1.2. Reflexión difusa 83
5.1.3. Reflexión especular 84
5.1.4. Atenuación 86
5.1.5. Materiales 87
5.1.6. El modelo de Phong 87
5.2. Tipos de fuentes de luz 89
5.3. Modelos de sombreado 91
5.4. Implementación de Phong con WebGL 93
5.4.1. Normales en los vértices 93
5.4.2. Materiales 95
5.4.3. Fuente de luz 96
5.5. Iluminación por ambas caras 98
5.6. Sombreado cómic 98
5.7. Niebla 100

«In trying to improve the quality of the synthetic images, we do not expect to be able to display the object exactly as it would appear in reality, with texture, overcast shadows, etc. We hope only to display an image that approximates the real object closely enough to provide a certain degree of realism».

Bui Tuong Phong, 1975

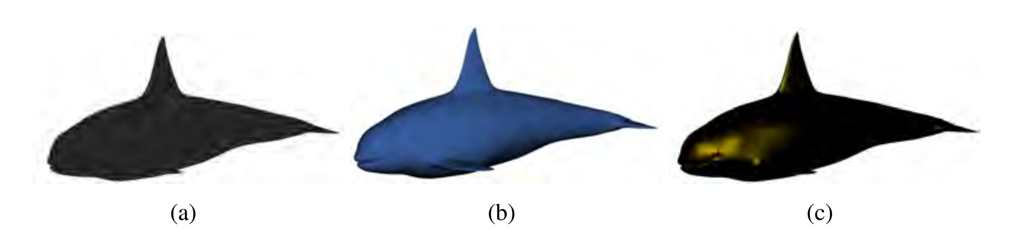
José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

5.1. Modelo de iluminación de Phong

El modelo de iluminación de Phong tiene en cuenta los tres aspectos siguientes:

- Luz ambiente: luz que proporciona iluminación uniforme a lo largo de la escena (véase figura 5.1(a)).
- Reflexión difusa: luz reflejada por la superficie en todas las direcciones (véase figura 5.1(b)).
- Reflexión especular: luz reflejada por la superficie en una sola dirección o en un rango de ángulos muy cercano al ángulo de reflexión perfecta (véase figura 5.1(c)).



(a) (b) (c)

Figura 5.1: Ejemplo de las componentes del modelo de iluminación de Phong: (a) Luz ambiente; (b) Reflexión difusa; (c) Reflexión especular

Y la combinación de estos tres aspectos produce el resultado que se muestra en la figura 5.2.

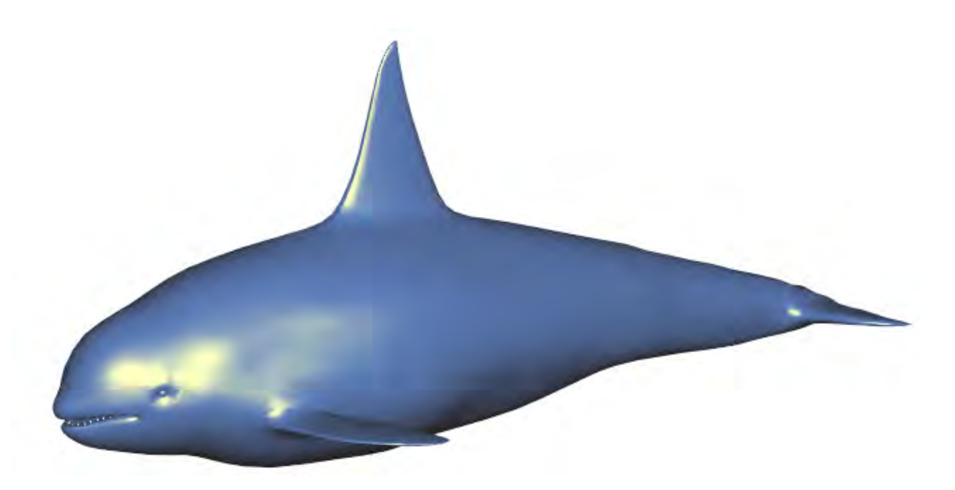


Figura 5.2: Ejemplo obtenido utilizando el modelo de iluminación de Phong

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

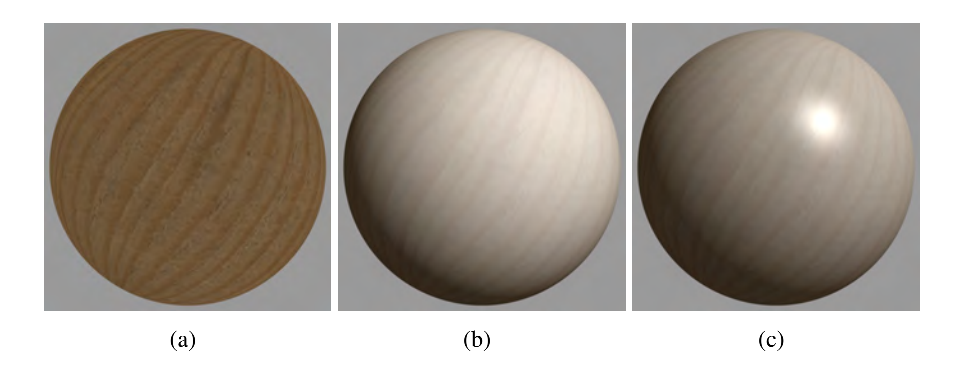
5.1.1. Luz ambiente

El modelo de Phong establece que la luz ambiente en una escena es constante. Esto produce que la iluminación que se observa en cualquier punto de la superficie de un objeto debida a la luz ambiente sea siempre la misma. Sin embargo, los diferentes objetos de la escena pueden estar construidos de diferentes materiales y cada material puede reflejar en mayor o menor medida la luz ambiente. El modelo de Phong modela este efecto con el coeficiente k_a, $0 \leq k_a \leq 1$, que será característico de cada material.

Si L_a es la luz ambiente, la iluminación ambiente I_a que se observa en un punto de la escena depende tanto de dicha luz como del material del objeto y se calcula de la siguiente manera:

$$I_a = k_a L_a \tag{5.1}$$

La figura 5.3(a) muestra un ejemplo en el que el modelo de iluminación únicamente incluye luz ambiente.



(a) (b) (c)

Figura 5.3: Ejemplos de iluminación: (a) Solo luz ambiente; (b) Luz ambiente y reflexión difusa; (c) Luz ambiente, reflexión difusa y reflexión especular

5.1.2. Reflexión difusa

La reflexión difusa es característica de superficies rugosas, mates, sin brillo. El modelo de Phong establece que para modelar este tipo de reflexión se utilice la ley de Lambert. Así, la iluminación observada en un punto de la superficie de un objeto depende del ángulo θ, $0 \leq \theta \leq 90$, entre la dirección a la fuente de luz L y la normal N de la superficie en dicho punto (véase figura 5.4). De manera similar a la luz ambiente, el modelo de Phong también tiene en cuenta que el material del objeto influye en la cantidad de luz reflejada y propone utilizar el coeficiente k_d, $0 \leq k_d \leq 1$, que también será propio del material.

Dada una fuente de luz, si L y N son vectores unitarios y L_d es la cantidad de luz producida por dicha fuente, la iluminación observada en un punto de la super-

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

ficie de un objeto debido a la reflexión difusa, I_d, se obtiene mediante la siguiente ecuación:

$$I_d = k_d L_d \cos\theta = k_d L_d (L \cdot N) \tag{5.2}$$

La figura 5.3(b) muestra un ejemplo en el que el modelo de iluminación incluye luz ambiente y reflexión difusa.

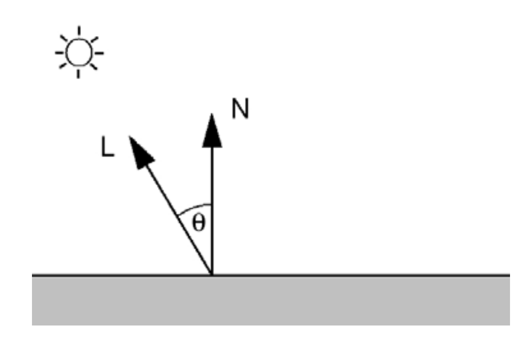


Figura 5.4: Vectores involucrados en el cálculo de la reflexión difusa

5.1.3. Reflexión especular

Este tipo de reflexión es propia de superficies brillantes, pulidas, y es responsable de los brillos que suelen observarse en esos tipos de superficies. Su principal característica es que este tipo de reflexión depende de la posición del observador. También ocurre que el color del brillo suele ser diferente del color de la superficie del objeto y más bien parecido al color de la luz cuya fuente es responsable del brillo.

El modelo de Phong establece que la luz que llega al observador depende del ángulo Φ entre el vector de reflexión perfecta R y el vector de dirección al observador V (véase figura 5.5). De manera similar a las otras dos componentes, el modelo de Phong también tiene en cuenta que el material del objeto influye en la cantidad de luz especular reflejada y utiliza el coeficiente k_s, $0 \leq k_s \leq 1$, que será propio del material. Además, propone modelar el tamaño del brillo, α, como exponente del producto escalar de los vectores R y V y que al igual que la k_d dependa del material del objeto.

Dada una fuente de luz, si R y V son vectores unitarios y L_s es la cantidad de luz producida por dicha fuente, la iluminación observada en un punto de la superficie de un objeto debido a la reflexión especular, I_s, se obtiene mediante la siguiente ecuación:

$$I_s = k_s L_s \cos^{\alpha}\Phi = k_s L_s (R \cdot V)^{\alpha} \tag{5.3}$$

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

donde R se obtiene de la siguiente manera:

$$R = 2N(N \cdot L) - L \tag{5.4}$$

Señalar que GLSL proporciona la función *reflect* que implementa el cálculo del vector R:

- `R = reflect (I, N);`

donde I es el vector incidente, es decir, $I = -L$.

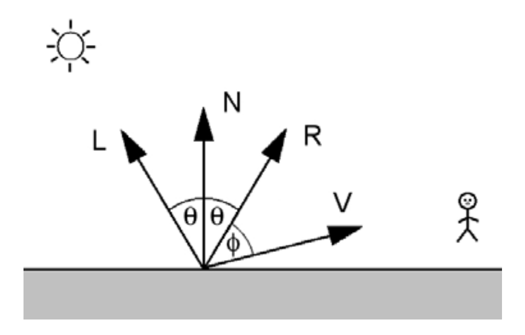
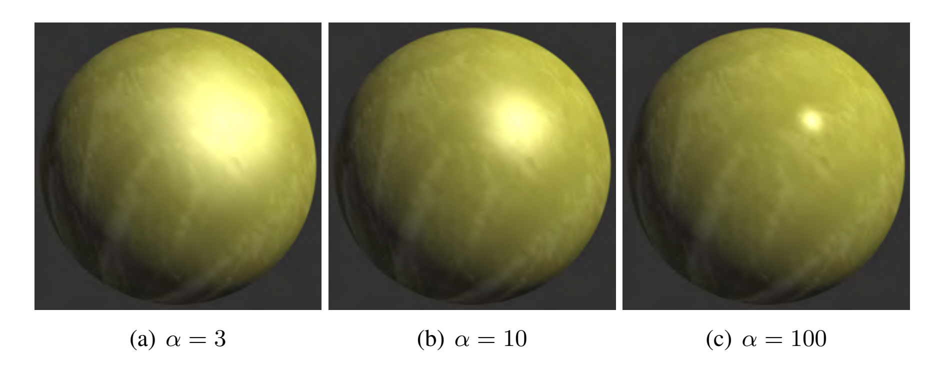


Figura 5.5: Vectores involucrados en el cálculo de la reflexión especular

La figura 5.3(c) muestra un ejemplo en el que el modelo de iluminación incluye luz ambiente, reflexión difusa y reflexión especular.

Respecto al valor de α, un valor igual a 1 modela un brillo grande, mientras que valores mucho mayores, por ejemplo, entre 100 y 500, modelan brillos más pequeños, propios de materiales, por ejemplo, metálicos. La figura 5.6 muestra varios ejemplos obtenidos con distintos valores de α.



(a) $\alpha = 3$ (b) $\alpha = 10$ (c) $\alpha = 100$

Figura 5.6: Ejemplos de iluminación con diferentes valores de α para el cálculo de la reflexión especular

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

El vector intermedio

El uso de la función `reflect` es caro. En su lugar, se propone a modo de optimización utilizar el vector intermedio, H. Este vector se calcula como la suma de los vectores L y V (véase figura 5.7). En el modelo de Phong, la reflexión especular se calcula ahora a partir del producto escalar entre el vector intermedio y la normal de la siguiente forma:

$$\begin{aligned} H &= L + V \\ I_s &= k_s L_s (N \cdot H)^\alpha \end{aligned} \tag{5.5}$$

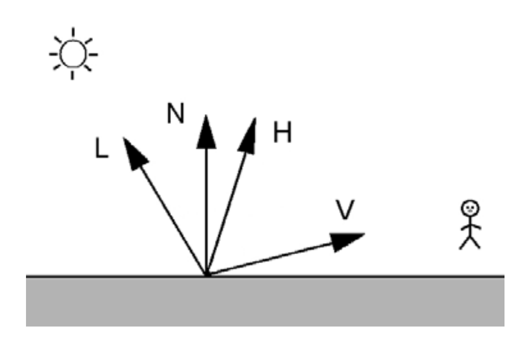


Figura 5.7: Vectores involucrados en el cálculo de la reflexión especular con el vector intermedio

5.1.4. Atenuación

Para tener en cuenta la atenuación que sufre la luz al viajar desde su fuente de origen hasta la superficie del objeto situado a una distancia d (véase figura 5.8), el modelo de Phong establece utilizar la siguiente ecuación donde los coeficientes a, b y c son constantes características de la fuente de luz:

$$attenuationFactor = \frac{1}{a + bd + cd^2} \tag{5.6}$$

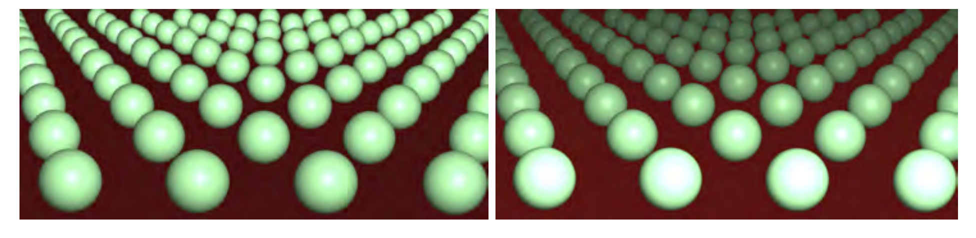


Figura 5.8: Ejemplo de escena sin atenuación (izquierda) y con atenuación de la luz (derecha)

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

5.1.5. Materiales

El modelo de iluminación de Phong tiene en cuenta las propiedades del material del objeto al calcular la iluminación y así proporcionar mayor realismo. En concreto son cuatro: ambiente k_a, difusa k_d, especular k_s y brillo α. La tabla 5.1 muestra una lista de materiales con los valores de ejemplo para estas constantes. Y la figura 5.9 muestra algunos resultados de su aplicación.

Esmeralda	”k_a” : [0.022, 0.17, 0.02] ”k_d” : [0.08, 0.61, 0.08] ”k_s” : [0.63, 0.73, 0.63] ”α” : [0.6]	Jade	”k_a” : [0.14, 0.22, 0.16] ”k_d” : [0.54, 0.89, 0.63] ”k_s” : [0.32, 0.32, 0.32] ”α” : [0.10]
Obsidiana	”k_a” : [0.05, 0.05, 0.07] ”k_d” : [0.18, 0.17, 0.23] ”k_s” : [0.33, 0.33, 0.35] ”α”: [0.30]	Perla	”k_a” : [0.25, 0.21, 0.21] ”k_d” : [1.0, 0.83, 0.83] ”k_s” : [0.30, 0.30, 0.30] ”α”: [0.09]
Rubí	”k_a” : [0.18, 0.01, 0.01] ”k_d” : [0.61, 0.04, 0.04] ”k_s” : [0.73, 0.63, 0.63] ”α”: [0.60]	Turquesa	”k_a” : [0.10, 0.19, 0.17] ”k_d” : [0.39, 0.74, 0.69] ”k_s” : [0.29, 0.31, 0.31] ”α”: [0.10]
Bronce	”k_a” : [0.21, 0.13, 0.05] ”k_d” : [0.71, 0.43, 0.18] ”k_s” : [0.39, 0.27, 0.17] ”α”: [0.20]	Cobre	”k_a” : [0.19, 0.07, 0.02] ”k_d” : [0.71, 0.27, 0.08] ”k_s” : [0.26, 0.14, 0.09] ”α”: [0.10]
Oro	”k_a” : [0.25, 0.20, 0.07] ”k_d” : [0.75, 0.61, 0.23] ”k_s” : [0.63, 0.56, 0.37] ”α”: [0.40]	Plata	”k_a” : [0.20, 0.20, 0.20] ”k_d” : [0.51, 0.51, 0.51] ”k_s” : [0.51, 0.51, 0.51] ”α”: [0.40]
Plástico	”k_a” : [0.0, 0.0, 0.0] ”k_d” : [0.55, 0.55, 0.55] ”k_s” : [0.70, 0.70, 0.70] ”α”: [0.25]	Goma	”k_a” : [0.05, 0.05, 0.05] ”k_d” : [0.50, 0.50, 0.50] ”k_s” : [0.70, 0.70, 0.70] ”α”: [0.08]

Tabla 5.1: Ejemplos de propiedades de algunos materiales para el modelo de Phong

5.1.6. El modelo de Phong

A partir de las ecuaciones 5.1, 5.2, 5.3 y 5.6 se define el modelo de iluminación de Phong como:

$$I = k_a L_a + \frac{1}{a + bd + cd^2}(k_d L_d (L \cdot N) + k_s L_s (R \cdot V)^\alpha) \qquad (5.7)$$

El listado 5.1 muestra la función que calcula la iluminación en un punto sin incluir el factor de atenuación. En el caso de tener múltiples fuentes de luz, hay que sumar los términos de cada una de ellas:

$$I = k_a L_a + \sum_{1 \le i \le m} \frac{1}{a_i + b_i d + c_i d^2}(k_d L_{d_i} (L_i \cdot N) + k_s L_{s_i} (R_i \cdot V)^\alpha) \qquad (5.8)$$

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

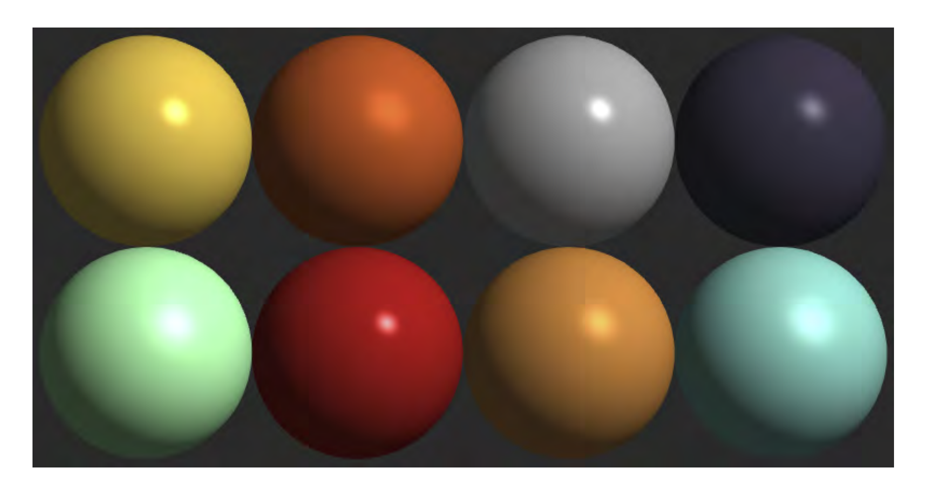


Figura 5.9: Ejemplos de materiales, de izquierda a derecha y de arriba a abajo: Oro, Cobre, Plata, Obsidiana, Jade, Rubí, Bronce, Turquesa.

Listado 5.1: Función que implementa para una fuente de luz el modelo de iluminación de Phong sin incluir el factor de atenuación

```
struct LightData {
    vec3 Position; // Posición en coordenadas del ojo
    vec3 La;       // Ambiente
    vec3 Ld;       // Difusa
    vec3 Ls;       // Especular
};
uniform LightData Light;

struct MaterialData {
    vec3 Ka;       // Ambiente
    vec3 Kd;       // Difusa
    vec3 Ks;       // Especular
    float alpha;   // Brillo especular
};
uniform MaterialData Material;

// N, L y V se asumen normalizados
vec3 phong (vec3 N, vec3 L, vec3 V) {

    vec3  ambient  = Material.Ka * Light.La;
    vec3  diffuse  = vec3(0.0);
    vec3  specular = vec3(0.0);
    float NdotL    = dot (N,L);

    if (NdotL > 0.0) {
      vec3  R       = reflect(-L, N);
      float RdotV_n = pow(max(0.0, dot(R,V)), Material.alpha);
      diffuse  = NdotL   * (Light.Ld * Material.Kd);
      specular = RdotV_n * (Light.Ls * Material.Ks);
    }
    return (ambient + diffuse + specular);
}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

▶ **5.1** Si deseas incorporar el factor de atenuación en la función del listado 5.1, ¿en qué estructura habría que incluir los parámetros a, b y c?
▶ **5.2** Indica los cambios que realizarías en la función del listado 5.1 para implementar la optimización que propone el método del vector intermedio.

5.2. Tipos de fuentes de luz

En general, siempre se han considerado dos tipos de fuentes de luz dependiendo de su posición:

- Posicional: la fuente emite luz en todas las direcciones desde un punto dado, muy parecido a como ilumina una bombilla, por ejemplo.
- Direccional: la fuente está ubicada en el infinito, todos los rayos de luz son paralelos y viajan en la misma dirección. En este caso el vector L en el modelo de iluminación de Phong es constante.

En ocasiones se desea restringir los efectos de una fuente de luz posicional a un área limitada de la escena, tal y como haría, por ejemplo, una linterna. A este tipo de fuente posicional se le denomina foco (véase figura 5.10).

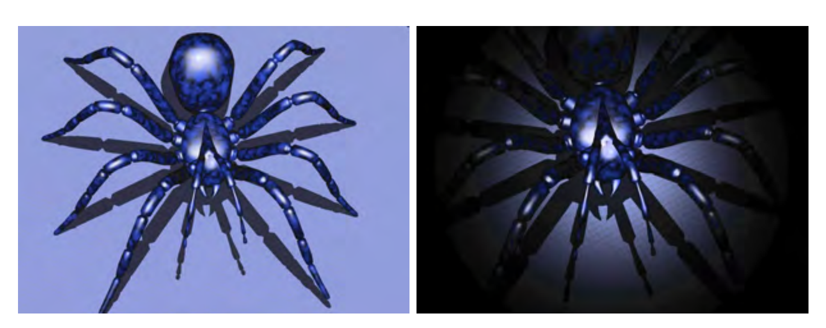


Figura 5.10: Ejemplo de escena iluminada: a la izquierda, con una luz posicional y a la derecha, con la fuente convertida en foco

A diferencia de una luz posicional, un foco viene dado, además de por su posición, por la dirección S y el ángulo δ que determinan la forma del cono, tal y como se muestra en la figura 5.11. Un fragmento es iluminado por el foco solo si está dentro del cono de luz. Esto se averigua calculando el ángulo entre el vector L y el vector S. Si el resultado es mayor que el ángulo δ es que ese fragmento está fuera del cono, siendo, en consecuencia, afectado únicamente por la luz ambiente.

Además, se puede considerar que la intensidad de la luz decae a medida que los rayos se separan del eje del cono. Esta atenuación se calcula mediante el coseno del ángulo entre los vectores L y S elevado a un exponente. Cuanto mayor sea este exponente, mayor será la concentración de luz alrededor del eje del cono (véase

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

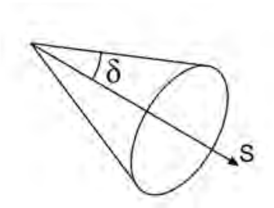


Figura 5.11: Parámetros característicos de un foco de luz

figura 5.10, imagen de la derecha). Finalmente, el factor de atenuación calculado se incorpora al modelo de iluminación de Phong, multiplicando el factor de atenuación que ya existía. El listado 5.2 muestra el nuevo *shader* que implementa el foco de luz (la estructura *LightData* muestra solo los campos nuevos).

Listado 5.2: *Shader* para calcular la iluminación con un foco de luz

```
struct LightData {     // Solo figuran los campos nuevos
    ....
    vec3  Direction; // Dirección de la luz en coordenadas del ojo
    float Exponent;  // Atenuación
    float Cutoff;    // Ángulo de corte en grados
}
uniform LightData Light;

vec3 phong (vec3 N, vec3 L, vec3 V) {

    vec3  ambient    = Material.Ka * Light.La;
    vec3  diffuse    = vec3(0.0);
    vec3  specular   = vec3(0.0);
    float NdotL      = dot (N,L);
    float spotFactor = 1.0;

    if (NdotL > 0.0) {

      vec3  S      = normalize (Light.Position - ec);
      float angle  = acos(dot(-S, Light.Direction));
      float cutoff = radians(clamp(Light.Cutoff, 0.0, 90.0));

      if (angle < cutoff) {

        spotFactor= pow (dot(-S,Light.Direction),Light.Exponent);

        vec3  R       = reflect(-L, N);
        float RdotV_n = pow(max(0.0, dot(R,V)), Material.alpha);

        diffuse  = NdotL   * (Light.Ld * Material.Kd);
        specular = RdotV_n * (Light.Ls * Material.Ks);
      }
    }

    return (ambient + spotFactor * (diffuse + specular));
}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

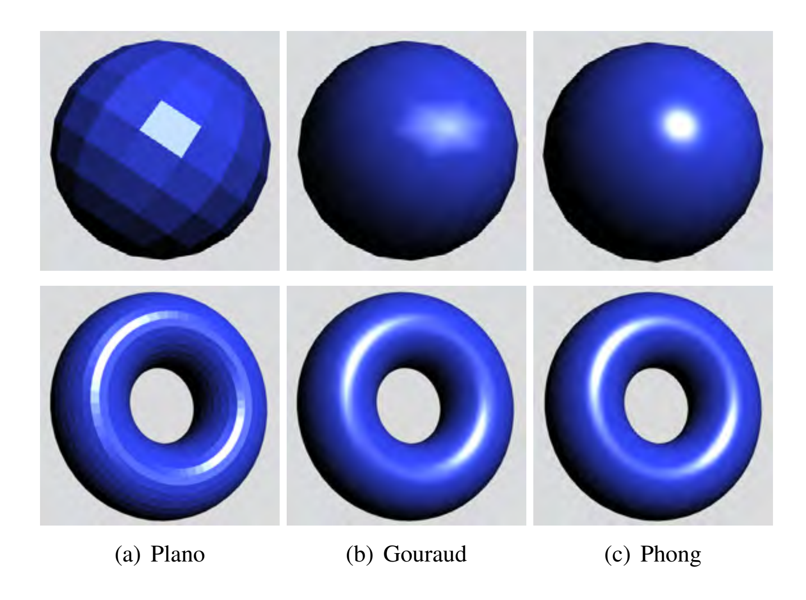
5.3. Modelos de sombreado

Un modelo de iluminación determina el color de la superficie en un punto. Un modelo de sombreado utiliza un modelo de iluminación y especifica cuándo usarlo. Dados un polígono y un modelo de iluminación, hay tres métodos para determinar el color de cada fragmento:

- Plano: el modelo de iluminación se aplica una sola vez y su resultado se aplica a toda la superficie del polígono. Este método requiere la normal de cada polígono. Un ejemplo del resultado obtenido se muestra en la figura 5.12(a). Para obtener este tipo de sombreado en WebGL 2.0 hay que utilizar el calificador *flat* con la variable que almacene el color (tanto en el *shader* de vértices como en el de fragmentos) para evitar que precisamente se realice la interpolación del color por fragmento.

- Gouraud: el modelo de iluminación se aplica en cada vértice del polígono y los resultados se interpolan sobre su superficie. Este método requiere la normal en cada uno de los vértices del polígono. Un ejemplo del resultado obtenido se muestra en la figura 5.12(b). El listado 5.3 muestra el *shader* correspondiente a este modelo de sombreado.

- Phong: el modelo de iluminación se aplica para cada fragmento. Este método requiere la normal en el fragmento, que se puede obtener por interpolación de las normales de los vértices. Un ejemplo del resultado obtenido se muestra en la figura 5.12(c). El listado 5.4 muestra el *shader* correspondiente a este modelo de sombreado.



(a) Plano (b) Gouraud (c) Phong

Figura 5.12: Resultados obtenidos utilizando los diferentes modelos de sombreado

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 5.3: *Shader* para realizar un sombreado de Gouraud

```
#version 300 es    // Shader de vértices
uniform   mat4 projectionMatrix;
uniform   mat4 modelViewMatrix;
uniform   mat3 normalMatrix;

in vec3 VertexPosition;
in vec3 VertexNormal;

out vec3 colorOut;

// ... aquí va el código del listado 5.1 ó 5.2

void main()  {

    vec3 N          = normalize(normalMatrix * VertexNormal);
    vec4 ecPosition= modelViewMatrix * vec4(VertexPosition, 1.0);
    vec3 ec         = vec3(ecPosition);
    vec3 L          = normalize(Light.Position - ec);
    vec3 V          = normalize(-ec);

    colorOut        = phong(N,L,V);
    gl_Position     = projectionMatrix * ecPosition;
}

#version 300 es    // Shader de fragmentos
precision mediump float;

in   vec3 colorOut;
out vec4 fragmentColor;

void main() {
    fragmentColor = vec4(colorOut,1);
}
```

Listado 5.4: *Shader* para realizar un sombreado de Phong

```
#version 300 es    // Shader de vértices
uniform   mat4 projectionMatrix, modelViewMatrix;
uniform   mat3 normalMatrix;

in vec3 VertexPosition;
in vec3 VertexNormal;

out vec3 N, ec;

void main()  {

    N                = normalize(normalMatrix * VertexNormal);
    vec4 ecPosition = modelViewMatrix * vec4(VertexPosition, 1.0);
    ec               = vec3(ecPosition);
    gl_Position      = projectionMatrix * ecPosition;
}
```

```
#version 300 es    // Shader de fragmentos
precision mediump float;

// ... aquí va el código del listado 5.1 ó 5.2

in  vec3 N, ec;
out vec4 fragmentColor;

void main() {

    vec3 n         = normalize(N);
    vec3 L         = normalize(Light.Position - ec);
    vec3 V         = normalize(-ec);
    fragmentColor = vec4(phong(n,L,V), 1.0);
}
```

5.4. Implementación de Phong con WebGL

5.4.1. Normales en los vértices

Para poder aplicar el modelo de iluminación de Phong, es necesario que se proporcionen las normales para cada vértice. Por ejemplo, el listado 5.5 define el modelo de un cubo donde para cada vértice se proporcionan la posición y la normal. Observa que en el modelo del cubo, la normal de un vértice es diferente según la cara que lo utilice. Por este motivo, la descripción del cubo ha pasado de 8 a 24 vértices.

Listado 5.5: Modelo de un cubo con la normal definida para cada vértice

```
var exampleCube = {

    "vertices" : [-0.5,-0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5,-0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5, 0.5, 0.5, 0.0, 0.0, 1.0,
                  -0.5, 0.5, 0.5, 0.0, 0.0, 1.0,
                   0.5,-0.5, 0.5, 1.0, 0.0, 0.0,
                   0.5,-0.5,-0.5, 1.0, 0.0, 0.0,
                   0.5, 0.5,-0.5, 1.0, 0.0, 0.0,
                   0.5, 0.5, 0.5, 1.0, 0.0, 0.0,
                   0.5,-0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5,-0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5, 0.5,-0.5, 0.0, 0.0,-1.0,
                   0.5, 0.5,-0.5, 0.0, 0.0,-1.0,
                  -0.5,-0.5,-0.5,-1.0, 0.0, 0.0,
                  -0.5,-0.5, 0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5, 0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5,-0.5,-1.0, 0.0, 0.0,
                  -0.5, 0.5, 0.5, 0.0, 1.0, 0.0,
                   0.5, 0.5, 0.5, 0.0, 1.0, 0.0,
                   0.5, 0.5,-0.5, 0.0, 1.0, 0.0,
                  -0.5, 0.5,-0.5, 0.0, 1.0, 0.0,
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

```
                    -0.5,-0.5,-0.5, 0.0,-1.0, 0.0,
                     0.5,-0.5,-0.5, 0.0,-1.0, 0.0,
                     0.5,-0.5, 0.5, 0.0,-1.0, 0.0,
                    -0.5,-0.5, 0.5, 0.0,-1.0, 0.0],

    "indices" : [ 0, 1, 2, 0, 2, 3, 4, 5, 6, 4, 6, 7,
                  8, 9,10, 8,10,11,12,13,14,12,14,15,
                 16,17,18,16,18,19,20,21,22,20,22,23]
};
```

La función *initShaders* es un buen sitio donde obtener la referencia al nuevo atributo y habilitarlo, así como para obtener la referencia a la matriz de transformación de la normal (véase listado 5.6).

Listado 5.6: Obtención de referencias para el uso de las normales

```
program.vertexNormalAttribute =
    gl.getAttribLocation ( program, "VertexNormal");
program.normalMatrixIndex     =
    gl.getUniformLocation( program, "normalMatrix");
gl.enableVertexAttribArray(program.vertexNormalAttribute);
```

Por supuesto, la matriz de la normal es propia de cada modelo, ya que se calcula a partir de la matriz modelo-vista y se obtiene, como se explicó en el capítulo 3, mediante la traspuesta de su inversa. Como esta operación hay que realizarla para cada modelo, es conveniente crear una función específica y llamarla antes de ordenar su dibujado para obtener así la matriz de la normal del *shader*, tal y como se muestra en el listado 5.7.

Listado 5.7: Funciones para el cálculo y la inicialización de la matriz de la normal en el *shader* a partir de la matriz modelo-vista

```
function getNormalMatrix(modelViewMatrix) {

  var normalMatrix = mat3.create();

  mat3.normalFromMat4(normalMatrix, modelViewMatrix);

  return normalMatrix;

}

function setShaderNormalMatrix(normalMatrix) {

  gl.uniformMatrix3fv(program.normalMatrixIndex, false,
      normalMatrix);

}
```

Por último, a la hora de dibujar el modelo, hay que especificar cómo se encuentra almacenado dicho atributo en el vector de vértices (véase listado 5.8).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 5.8: Nueva función de dibujo que incluye dos atributos: posición y normal

```
function drawSolid(model) {

  gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
  gl.vertexAttribPointer (program.vertexPositionAttribute, 3,
                          gl.FLOAT, false, 2*3*4,   0);
  gl.vertexAttribPointer (program.vertexNormalAttribute,   3,
                          gl.FLOAT, false, 2*3*4, 3*4);

  gl.bindBuffer   (gl.ELEMENT_ARRAY_BUFFER,
                   model.idBufferIndices);
  gl.drawElements (gl.TRIANGLES, model.indices.length,
                   gl.UNSIGNED_SHORT, 0);

}
```

5.4.2. Materiales

Para especificar un material es necesario, en primer lugar, obtener las referencias de las variables que van a contener el valor del material en el *shader*. Examina el listado 5.1 para recordarlas. El listado 5.9 muestra el código correspondiente a este paso. Un buen sitio para colocarlo sería en la función *initShaders*.

Listado 5.9: Obtención de las referencias a las variables del *shader* que contendrán el material

```
program.KaIndex    = gl.getUniformLocation(program, "Material.Ka");
program.KdIndex    = gl.getUniformLocation(program, "Material.Kd");
program.KsIndex    = gl.getUniformLocation(program, "Material.Ks");
program.alphaIndex= gl.getUniformLocation(program,
                     "Material.alpha");
```

En segundo lugar, hay que especificar el material para cada modelo justo antes de ordenar su dibujado. Para esto, es buena idea definir una función que inicialice las variables del *shader* correspondientes al material. El listado 5.10 muestra un ejemplo del material *Silver*, una función para asignar valores de material a las variables del *shader* así como un ejemplo de cómo dibujar un cubo con dicho material.

Listado 5.10: La función *setShaderMaterial* recibe un material como parámetro e inicializa las variables del *shader* correspondientes. En la función *drawScene* se establece un valor de material antes de dibujar el objeto

```
var Silver = {

  "mat_ambient" : [ 0.19225, 0.19225, 0.19225 ],
  "mat_diffuse" : [ 0.50754, 0.50754, 0.50754 ],
  "mat_specular": [ 0.50827, 0.50827, 0.50827 ],
  "alpha"       : [ 51.2 ]

};
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

```
function setShaderMaterial(material) {

  gl.uniform3fv(program.KaIndex,     material.mat_ambient);
  gl.uniform3fv(program.KdIndex,     material.mat_diffuse);
  gl.uniform3fv(program.KsIndex,     material.mat_specular);
  gl.uniform1f (program.alphaIndex,  material.alpha);
}

function drawScene () {

  ....
  // establece un material y dibuja un cubo
  setShaderMaterial(Silver);
  drawSolid(exampleCube);
  ....
}
```

5.4.3. Fuente de luz

Respecto a la fuente de luz, por una parte hay que obtener las referencias a las variables del *shader* (véase listado 5.11) que definen los parámetros de la fuente de luz, y que de manera similar al material podemos colocar en la función *initShaders*. Por otra parte, hay que inicializar las variables del *shader* antes de ordenar el dibujado del primer objeto de la escena. Es interesante agrupar la inicialización en una sola función (véase listado 5.12).

Listado 5.11: Obtención de las referencias a las variables del *shader* que contendrán los valores de la fuente de luz

```
program.LaIndex       = gl.getUniformLocation(program, "Light.La");
program.LdIndex       = gl.getUniformLocation(program, "Light.Ld");
program.LsIndex       = gl.getUniformLocation(program, "Light.Ls");
program.PositionIndex= gl.getUniformLocation(program,
                        "Light.Position");
```

Listado 5.12: La función *setShaderLight* inicializa las variables del *shader* correspondientes

```
function setShaderLight() {

  gl.uniform3f(program.LaIndex,          1.0,   1.0,  1.0);
  gl.uniform3f(program.LdIndex,          1.0,   1.0,  1.0);
  gl.uniform3f(program.LsIndex,          1.0,   1.0,  1.0);
  gl.uniform3f(program.PositionIndex,  10.0,  10.0,  0.0);

}
```

Ejercicios

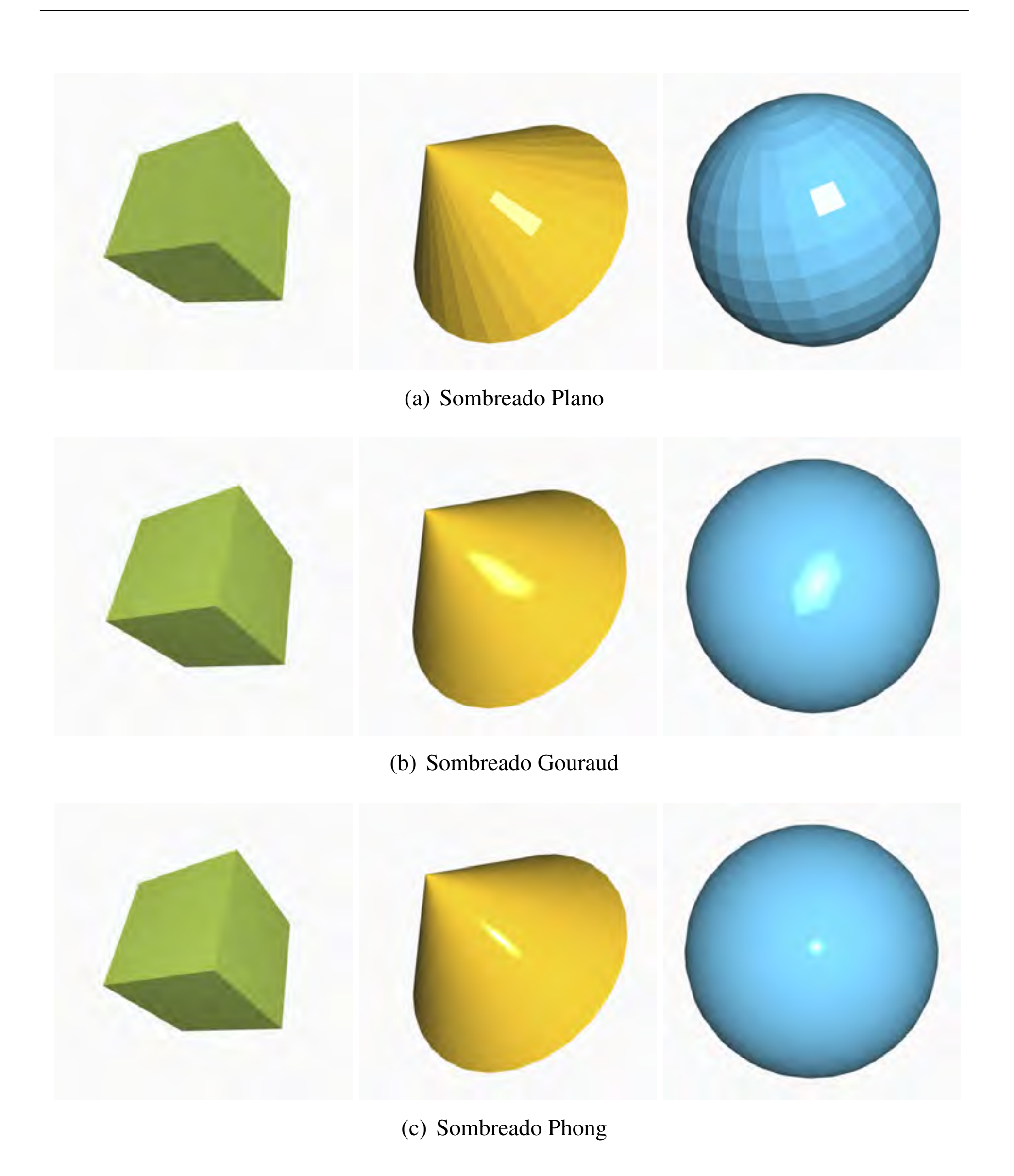
▶ **5.3** Ejecuta el programa *c05/phongConSombreadoGouraud.html*, que implementa el modelo de iluminación de Phong y el modelo de sombreado de Gouraud. Como resultado obtendrás imágenes similares a las de la figura 5.13(b). Examina el *shader* y comprueba

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

que los fragmentos de código comentados en esta sección se incluyen en *c05/iluminacion.js*. Comprueba también cómo el fichero *c05/primitivasGN.js* contiene las normales de cada vértice para cada una de las primitivas definidas.

▶ **5.4** Implementa el modelo de sombreado de Phong a partir del mismo programa del ejercicio anterior. La figura 5.13(c) muestra tres ejemplos de resultados obtenidos utilizando este modelo de sombreado.

▶ **5.5** A partir del programa *c05/phongConSombreadoGouraud.html* realiza los cambios necesarios para implementar el modelo de sombreado plano. La figura 5.13(a) muestra algunos resultados obtenidos utilizando este modelo de sombreado.



(a) Sombreado Plano

(b) Sombreado Gouraud

(c) Sombreado Phong

Figura 5.13: Ejemplos obtenidos con los modelos de sombreado: Plano, Gouraud y Phong

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

5.5. Iluminación por ambas caras

Cuando la escena incorpora modelos abiertos, es posible observar los polígonos que se encuentran en la parte trasera de los objetos, como, por ejemplo, ocurre en la copa de la figura 5.14. Para una correcta visualización es necesario utilizar la normal contraria en los polígonos que forman parte de la cara trasera. Esto es una operación muy sencilla en WebGL y se muestra en el listado 5.13.

Listado 5.13: Iluminación en ambas caras, modificación en el *shader* de fragmentos

```
if (gl_FrontFacing)
    fragmentColor = vec4(phong( n, L, V), 1.0);
else
    fragmentColor = vec4(phong(-n, L, V), 1.0);
```

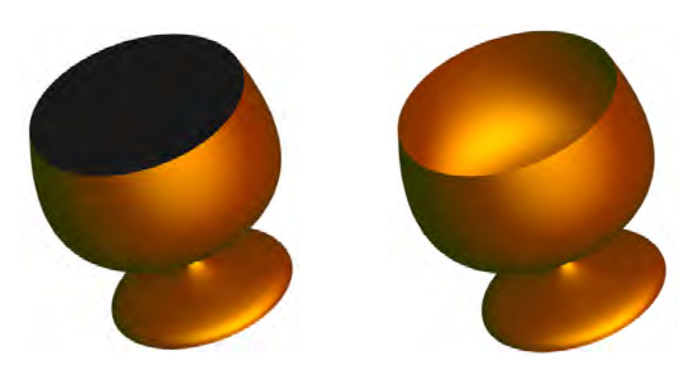


Figura 5.14: Ejemplo de modelo en el que hay que aplicar iluminación en ambas caras para una correcta visualización

Ejercicios

▶ **5.6** Implementa un *shader* que te permita sombrear objetos por ambas caras. Esto es muy útil en el caso de que los objetos sean abiertos como, por ejemplo, ocurre con las primitivas del cilindro, el cono o el plano. El listado 5.13 muestra el código necesario. Decide tú mismo dónde ponerlo (quizá debas también eliminar alguna línea en tu *shader* actual). Recuerda que solo verás diferencias si alguna de las primitivas de la escena es abierta.

▶ **5.7** Tratar las dos caras de los polígonos te permitiría utilizar un tipo de material diferente para cada cara. Por ejemplo, material oro para las caras delanteras y material plata para las traseras. Comenta las modificaciones que tendrías que hacer al *shader* del listado 5.1 para conseguirlo.

5.6. Sombreado cómic

El objetivo es simular el sombreado típico en cómics. Este efecto se consigue haciendo que la componente difusa del color de un fragmento se restrinja a solo un número determinado de posibles valores. La función *toonShading* que se muestra

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

en el listado 5.14 realiza esta operación para cada fragmento. La variable *levels* determina el número máximo de colores distintos (véase figura 5.15).

Listado 5.14: Iluminación en ambas caras, modificación en el *shader* de fragmentos

```
vec3 toonShading (vec3 N, vec3 L) {

    vec3  ambient      = Material.Ka * Light.La;
    float NdotL        = max(0.0, dot (N,L));
    float levels       = 3.0;
    float scaleFactor = 1.0 / levels;

    vec3 diffuse   = ceil(NdotL * levels) * scaleFactor *
                     (Light.Ld * Material.Kd);

    return (ambient + diffuse);

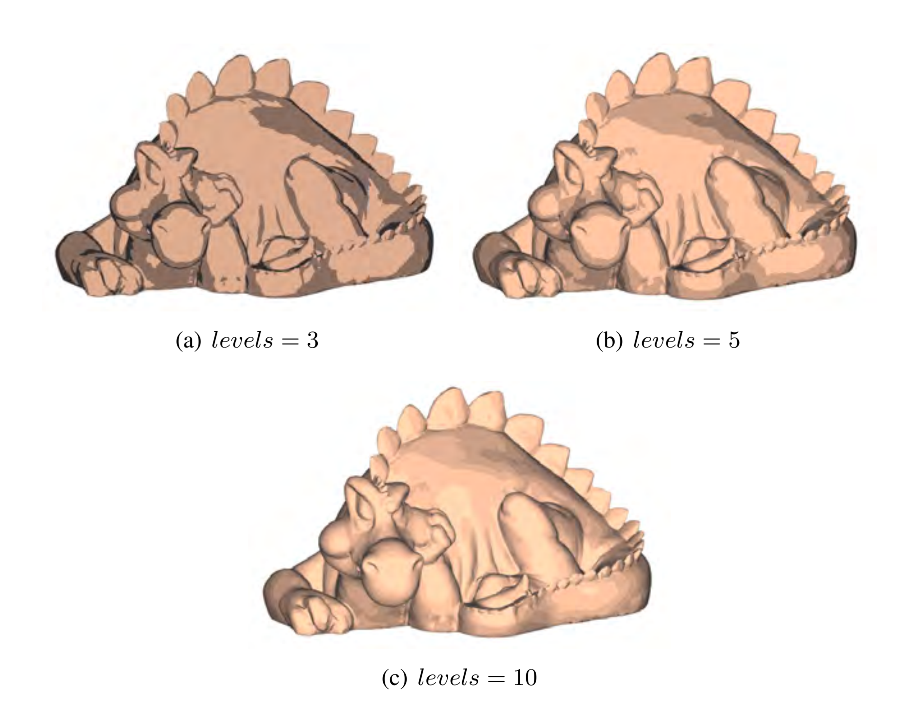
  }

void main() {

    vec3 n = normalize(N);
    vec3 L = normalize(Light.Position - ec);

    fragmentColor = vec4(toonShading(n,L), 1.0);

}
```



(a) $levels = 3$ (b) $levels = 5$

(c) $levels = 10$

Figura 5.15: Resultado de la función *toonShading* con diferentes valores de la variable *levels*

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

► **5.8** Añade la función *toonShading* a tu *shader* de fragmentos y haz que se llame a esta en lugar de a la función *phong*. Observa que la componente especular se ha eliminado de la ecuación, por lo que la función *toonShading* solo necesita los vectores N y L.

► **5.9** Para complementar el *shader* de sombreado cómic, sería muy interesante reforzar también los bordes del objeto dibujado utilizando el color negro. Dirígete a la bibliografía y encuentra técnicas que te permitan conseguir este segundo efecto utilizando *shaders*.

5.7. Niebla

El efecto de niebla se consigue mezclando el color de cada fragmento con el color de la niebla. A mayor distancia de un fragmento respecto a la cámara, mayor peso tiene el color de la niebla y al contrario, a menor distancia, mayor peso tiene el color del fragmento.

En primer lugar hay que obtener el color del fragmento y después, a modo de posproceso, se mezcla con el color de la niebla dependiendo de la distancia a la cámara. Para implementar este efecto se definen tres variables: distancia mínima, distancia máxima y color de la niebla. Así, dado un fragmento, tenemos tres posibilidades:

- Si el fragmento está a una distancia menor que la distancia mínima, no se ve afectado por la niebla, el color definitivo es el color del fragmento.
- Si el fragmento está a una distancia mayor que la máxima, el color definitivo es el color de la niebla (es importante que el color de fondo coincida con el de la niebla).
- Si el fragmento está a una distancia entre la mínima y la máxima, su color definitivo depende del color del fragmento y del de la niebla. Esta variación puede ser lineal con la distancia, pero suele producir mucho mejor resultado utilizar una función exponencial.

El listado 5.15 muestra la estructura *FogData* y el cálculo del valor de niebla de manera lineal y, con comentarios, de manera exponencial. La figura 5.16 muestra algunos resultados.

Listado 5.15: *Shader* de niebla

```
struct FogData {

    float maxDist;
    float minDist;
    vec3  color;

};
FogData Fog;
```


ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

```
void main() {

    Fog.minDist = 10.0;
    Fog.maxDist = 20.0;
    Fog.color   = vec3(0.15, 0.15, 0.15);

    vec3 n = normalize(N);
    vec3 L = normalize(Light.Position - ec);
    vec3 V = normalize(-ec);

    float dist      = abs (ec.z);

    // lineal
    loat fogFactor = (Fog.maxDist - dist) /
                     (Fog.maxDist - Fog.minDist);

    // exponencial
    // float fogFactor = exp(-pow(dist, 2.0));

    fogFactor        = clamp (fogFactor, 0.0, 1.0);
    vec3 phongColor = phong(n,L,V);
    vec3 myColor     = mix (Fog.color, phongColor, fogFactor);

    fragmentColor    = vec4(myColor, 1.0);

}
```

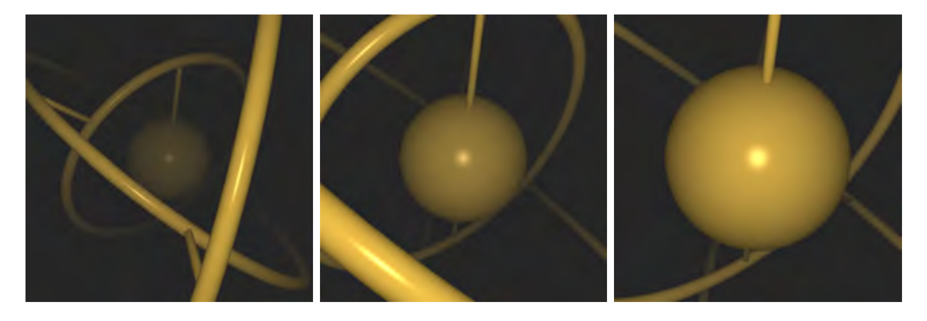


Figura 5.16: Resultados obtenidos utilizando el *shader* de niebla

Ejercicios

▶ **5.10** Implementa un *shader* que simule el efecto de niebla mezclando el color de cada fragmento con el color de la niebla dependiendo de la distancia del fragmento a la cámara (es importante que el color de fondo coincida con el color de la niebla). Utiliza el código del listado 5.15.

▶ **5.11** Busca la diferencia visual que produce la variación de la niebla si se calcula de manera lineal con la distancia o si se realiza de manera exponencial, ¿cuál te ha gustado más?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

CUESTIONES

▶ **5.1** ¿En qué *shader* se declara el atributo de la normal, vértices o fragmentos? ¿Por qué?

▶ **5.2** Respecto a la matriz de transformación de la normal, ¿cuál de las transformaciones básicas que conoces nunca está incluida en dicha matriz: traslación, giro o escalado?

▶ **5.3** ¿Por qué la matriz de transformación de la normal se suele obtener a partir de la matriz modelo-vista y no únicamente a partir de la matriz de transformación del modelo?

▶ **5.4** ¿Qué crees que pasaría con una normal en el caso de que la matriz de transformación del modelo contuviese una operación de escalado en el que uno de sus factores de escala estuviese a cero?

▶ **5.5** Hemos organizado los atributos de un vértice de tal forma que en primer lugar figura la posición y después la normal. ¿Crees que podríamos hacerlo al revés, es decir, primero la normal y después la posición?

▶ **5.6** ¿Qué representa la K_d para el objeto? ¿Y la K_s? ¿Podrías conseguir que una luz blanca ilumine una esfera blanca y que el brillo observado se aprecie de color azúl?

▶ **5.7** ¿A qué componentes del modelo del modelo de iluminación de Phong afecta el factor de atenuación de la luz?, ¿cómo lo aplicarías en la función `phong`?

▶ **5.8** ¿Qué es la variable `ec` que aparece en el *shader* de vértices que realiza un sombreado de Gouraud? ¿Y el vector V? ¿Sabrías dibujar un boceto donde se muestre lo que ambas variables representan?

▶ **5.9** Imagina que la fuente de luz está colocada en un punto fijo de la escena (como una lámpara, por ejemplo), y que la variable `Lp` contiene dicha posición, ¿qué tendrías que hacer para obtener la correcta posición de la luz que te permite seguir utilizando el *shader* del listado 5.3?

▶ **5.10** Explica las diferencias, ventajas e inconvenientes, que hay entre los tres métodos de sombreado: Plano, Gouraud y Phong.

▶ **5.11** En la implementación del modelo de sombreado de Phong, ¿en qué *shader* se calcula el vector V?

▶ **5.12** Si utilizas el modelo de iluminación de Phong con una fuente de luz direccional, ¿es cierto que al ser el vector L constante para toda la escena también lo será el vector de reflexión R?

▶ **5.13** En el modelo de iluminación de Phong, ¿qué significa que el producto escalar entre los vectores N y L sea negativo?

▶ **5.14** Si M es la matriz modelo, C es la de la cámara, P la de proyección, vp es el atributo de posición del vértice, y L_p es la posición de la fuente de luz, escribe el fragmento de código de un *shader* que calcula los vectores L y V para utilizarlos en el modelo de iluminación de Phong. Asume vp y Lp de tipo $vec3$.

▶ **5.15** En el modelo de iluminación de Phong, explica cómo se modela la reflexión especular propia de superficies brillantes o muy pulidas. Escribe el código GLSL que permite calcularla para un vértice del cual conoces su posición y normal, y también conoces la posición de la fuente de luz.

▶ **5.16** Según el modelo de iluminación de Phong, ¿qué información necesitas para poder calcular el color de un punto debido a la reflexión difusa? Escribe la expresión que te

permite obtenerlo para una fuente de luz posicional. ¿Qué diferencias hay si la fuente de luz fuese direccional?

► **5.17** En la implementación del modelo de sombreado de Gouraud, ¿qué información ha de interpolar la GPU?

► **5.18** Tu escena se compone únicamente de un cuadrado y utilizas el modelo de iluminación de Phong. Compara estas dos posibles situaciones: a) utilizar una fuente de luz posicional y sombreado plano; b) utilizar una fuente de luz direccional y sombreado de Gouraud. ¿Crees que en ambos casos se obtendrá un efecto de sombreado similar o si por el contrario habrá diferencias evidentes?

► **5.19** Explica la diferencia entre los conceptos modelo de iluminación de Phong y modelo de sombreado de Phong.

► **5.20** Dado una luz de tipo foco, ¿qué vectores y cómo se utilizan para saber si un fragmento está dentro del cono de luz?

► **5.21** ¿Qué tipo de sombreado (Plano, Gouraud o Phong) sería más conveniente utilizar en el caso de que tu escena mostrara objetos de superficie plana y de material especular (como un mesa de mármol muy pulida, por ejemplo)?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 6

Texturas

Índice

6.1. Coordenadas de textura **105**
6.1.1. Repetición de la textura 109
6.2. Leyendo téxeles **111**
6.2.1. Magnificación 112
6.2.2. Minimización 112
6.3. Texturas en WebGL **113**
6.4. Texturas 3D **116**
6.5. Mapas de cubo **117**
6.5.1. *Reflection mapping* 117
6.5.2. *Refraction mapping* 119
6.5.3. *Skybox* 119
6.6. *Normal mapping* **123**
6.7. *Displacement mapping* **123**
6.8. *Alpha mapping* **124**

En el capítulo anterior se mostró cómo un modelo de iluminación aumentaba el realismo visual de las imágenes generadas por computador. Ahora, el objetivo es mejorar un poco más dicho realismo visual mediante el uso de texturas, es decir, imágenes que a modo de mapas de color se utilizarán para calcular el color definitivo de los píxeles (observa las figuras 6.1 y 6.2). El uso de texturas para incrementar el realismo visual es muy frecuente, por lo que el número de técnicas en la literatura es también muy elevado. En concreto, en este capítulo se van a revisar técnicas que resultan especialmente idóneas para gráficos en tiempo real.

6.1. Coordenadas de textura

Las coordenadas de textura son necesarias para establecer cómo se ha de pegar la textura sobre la superficie del modelo. Por ejemplo, la figura 6.3 muestra cómo

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

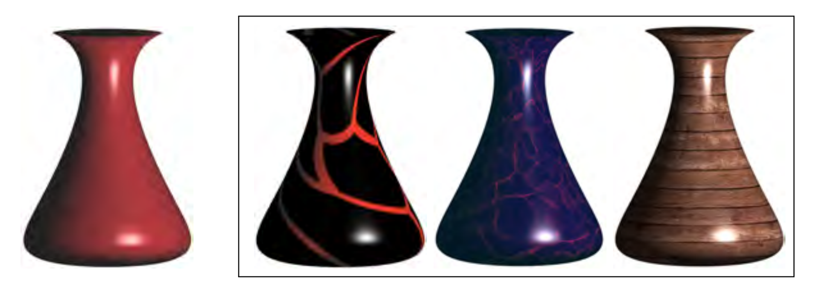


Figura 6.1: Resultados obtenidos al aplicar diferentes texturas 2D sobre el mismo objeto 3D

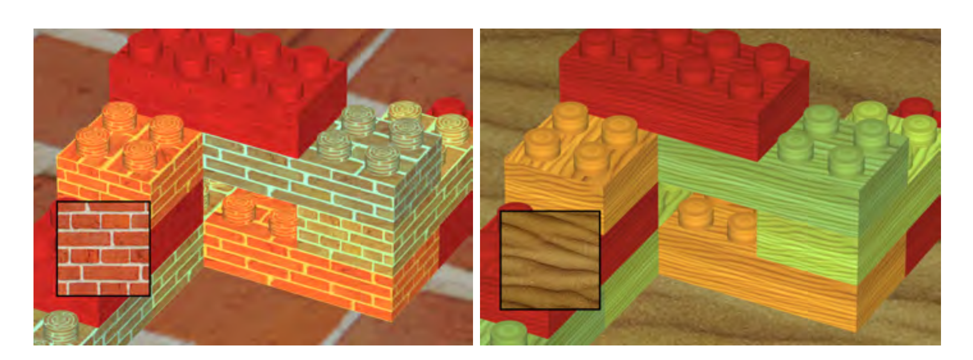


Figura 6.2: Ejemplo de aplicar una textura 2D diferente a la misma escena

una misma textura se ha aplicado sobre el mismo modelo produciendo resultados distintos. La diferencia reside únicamente en las coordenadas de textura que se han utilizado.

Las coordenadas de textura se suministran como un nuevo atributo del vértice, aunque lo más habitual es que sea en el procesador de fragmentos donde se accede a la textura por lo que será necesario que la GPU las interpole y así obtenerlas para cada fragmento. El rango de coordenadas válido en el espacio de la textura es $[0, 1]$, siendo este rango independiente del tamaño en píxeles de la textura (véanse figura 6.4).



Figura 6.3: Resultados obtenidos utilizando diferentes coordenadas de textura

El listado 6.1 muestra los cambios necesarios. El *shader* de vértices recibe el nuevo atributo que contiene las coordenadas de textura y las asigna a una variable de tipo *out* para que cada fragmento reciba sus coordenadas de textura interpoladas.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

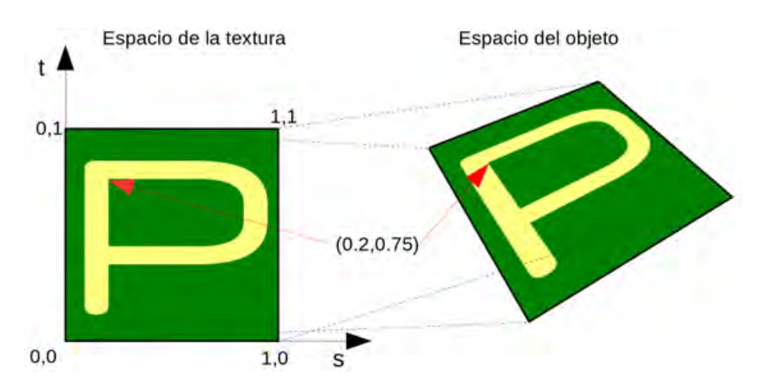


Figura 6.4: En el espacio de la textura, el rango de coordenadas válido es $[0, 1]$. En el espacio del objeto, cada fragmento recibe las coordenadas de textura interpoladas

El *shader* de fragmentos utiliza la función *texture* para, a partir de las coordenadas y una textura, obtener un valor de color. La textura está declarada de tipo *sampler2D*, que es el tipo que GLSL establece para una textura 2D. La figura 6.5 muestra dos resultados de la aplicación de textura.

Listado 6.1: Cambios necesarios para que un *shader* utilice una textura 2D

```
// Shader de vértices
...
in  vec2 VertexTexcoords;  // nuevo atributo
out vec2 texCoords;

void main()  {
  ...
  // se asignan las coordenadas de textura del vértice
  // a la variable texCoords
  texCoords = VertexTexcoords;

}

// Shader de fragmentos
...
uniform sampler2D myTexture;    // la textura
in vec2 texCoords;              // coords de textura interpoladas

void main() {
  ...
  // acceso a la textura para obtener un valor de color RGBA
  fragmentColor = texture(myTexture, texCoords);

}
```

Por supuesto, desde el *shader* de fragmentos es posible acceder a diferentes texturas y realizar cualquier combinación con los valores obtenidos para determinar el color definitivo. La figura 6.6 muestra un ejemplo donde se han combinado dos texturas y el listado 6.2 muestra los cambios necesarios en el *shader* de fragmentos

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

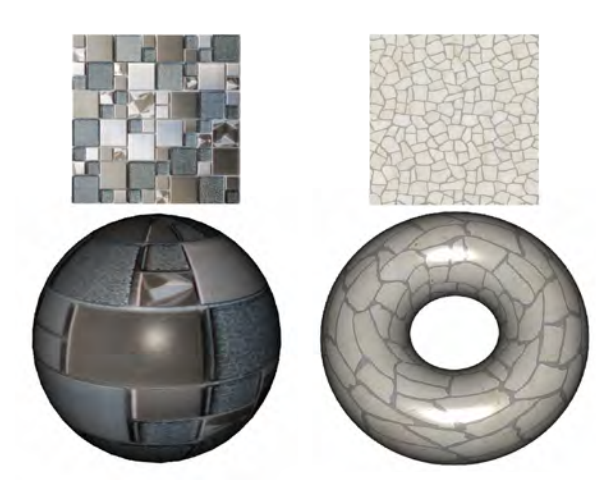


Figura 6.5: Ejemplos de aplicación de textura 2D. En estos casos el color definitivo de un fragmento se ha obtenido a partir de la textura y del modelo de iluminación de Phong

para acceder a varias texturas (el *shader* de vértices no cambia). Observa, por ejemplo, que se utilizan las mismas coordenadas de textura para acceder a las diferentes texturas. Observa también que se ha utilizado la operación de multiplicación para combinar los colores obtenidos de cada textura, pero se podría haber usado cualquier otra operación.

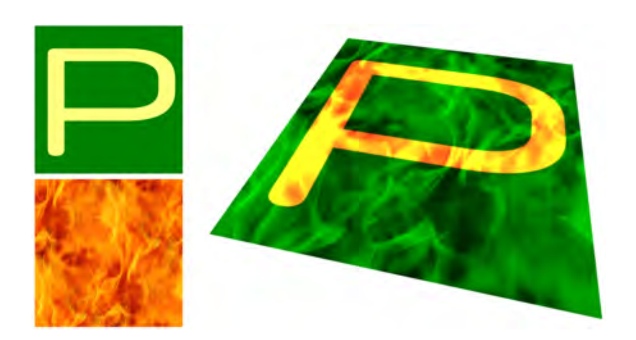


Figura 6.6: Resultado de combinar dos texturas

Listado 6.2: Cambios en el *shader* de fragmentos para utilizar varias texturas 2D

```
// Shader de fragmentos
...
uniform sampler2D myTexture1, myTexture2; // las texturas
in vec2 texCoords;                 // coords de textura interpoladas

void main() {
  ...
  // acceso a las texturas para obtener los valores de color RGBA
  fragmentColor = texture(myTexture1, texCoords) *
                  texture(myTexture2, texCoords);
}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

6.1.1. Repetición de la textura

En el caso de proporcionar valores mayores que 1 en las coordenadas de textura, es posible gobernar el resultado de la visualización utilizando la función *gl.texParameter*. Tres son los modos posibles, y pueden combinarse con valores distintos en cada dirección S y T (véanse ejes en la figura 6.4). Son estos:

- Repite la textura (véase figura 6.7)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.REPEAT);`
- Extiende el borde de la textura (véase figura 6.8)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.CLAMP_TO_EDGE);`
- Repite la textura de manera simétrica (véase figura 6.9)
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_[S|T], gl.MIRRORED_REPEAT);`

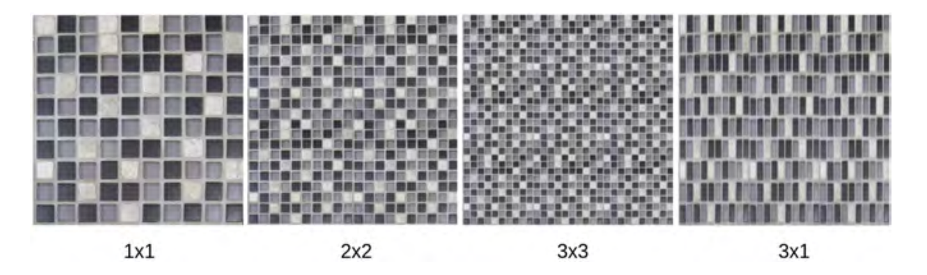


Figura 6.7: Ejemplos de repetición de textura

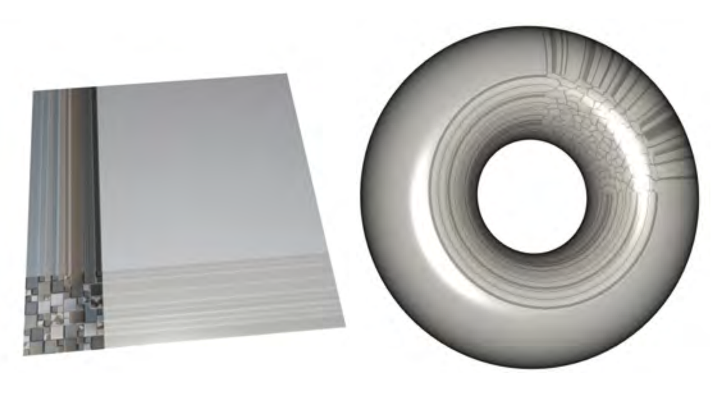


Figura 6.8: Resultados de extensión del borde de la textura

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

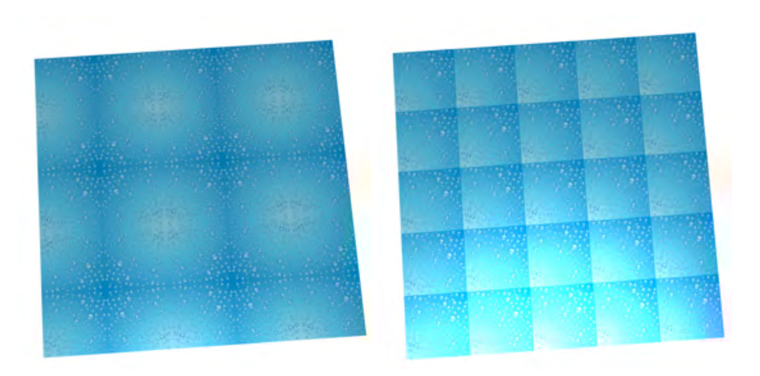
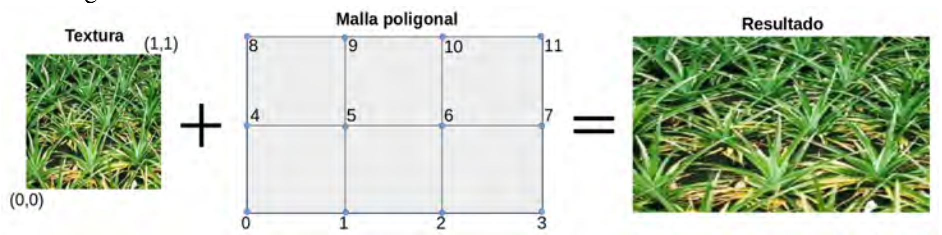


Figura 6.9: Repetición de la textura de manera simétrica (imagen de la izquierda) y repetición normal como la de la figura 6.7 (imagen de la derecha)

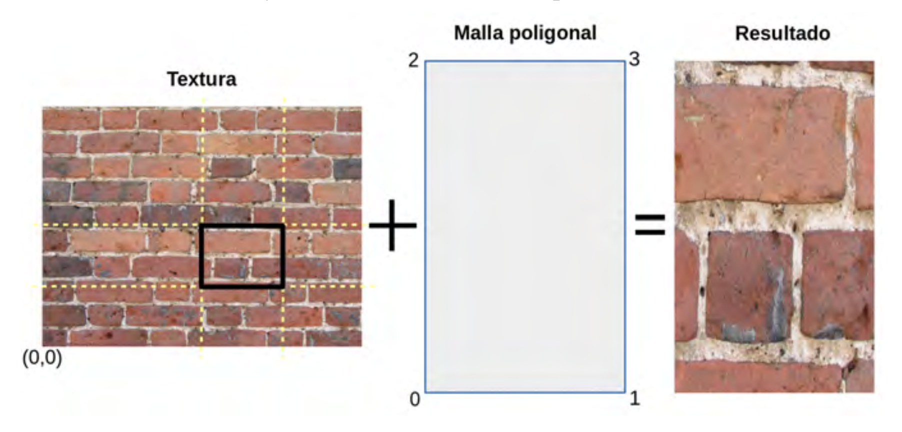
Ejercicios

► **6.1** Viendo la imagen izquierda de la figura 6.8, ¿cuáles crees que son las coordenadas de textura que se han especificado para cada uno de los vértices del polígono?, ¿y en el caso de la imagen izquierda de la figura 6.9?

► **6.2** Determina las coordenadas de textura de cada uno de los vértices que se encuentran numerados en la siguiente figura de manera que tras aplicar la textura de cesped se obtiene el siguiente resultado.

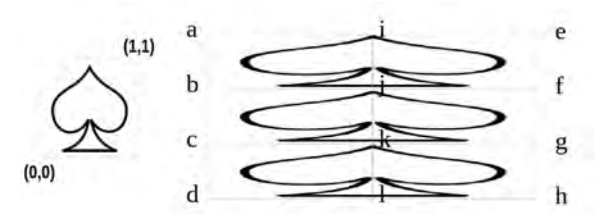


► **6.3** Determina las coordenadas de textura de cada uno de los vértices que se encuentran numerados en la siguiente figura. Para averiguarlas, ten en cuenta el resultado que también se muestra en la figura. En este caso, el ancho y alto del trozo de textura utilizado son de un cuarto del ancho y alto total de la textura respectivamente.



ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

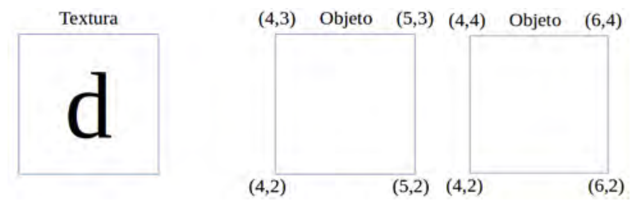
▶ **6.4** Determina las coordenadas de textura de los siguientes vértices (enumerados como a, b, c, d, e, f, g, h, i, j, k, l) de manera que la textura quede como se muestra en la siguiente figura.



▶ **6.5** Trabajando con coordenadas de textura mayores que uno y si se extiende el color del borde de la textura, ¿en qué coordenadas de textura se convertirán las coordenadas $(0.7, 2.3)$?

▶ **6.6** Trabajando con coordenadas de textura mayores a uno y si se repite la textura, ¿en qué coordenadas de textura se transforman las coordenadas $(3.5, 2.7)$? ¿y si las coordenadas son $(0.1, 7.7)$?

▶ **6.7** Utilizando la textura que se muestra en la siguiente figura y aplicando el modo de repetición basado en espejo de OpenGL (`MIRRORED_REPEAT`), dibuja como quedaría la textura sobre cada uno de los siguientes objetos teniendo en cuenta las coordenadas de textura que se especifican para cada uno de ellos.



▶ **6.8** Si para un fragmento sus coordenadas de textura son $(1.3, 0.7)$, obtén las respectivas coordenadas de textura en el espacio de la textura para cada uno de los métodos de repetición soportados en WebGL.

6.2. Leyendo téxeles

Un téxel es un píxel de la textura. En los orígenes de OpenGL, el acceso a los téxeles de una textura solo era posible realizarlo desde el *shader* de fragmentos. Sin embargo, los procesadores gráficos que aparecieron en el 2008 eliminaban esa limitación y era posible acceder a una textura también desde el *shader* de vértices. En cualquier caso, debido a que rara vez el tamaño de un fragmento se corresponde con el de un téxel de la textura, el acceso a la textura conlleva dos problemas:

- Magnificación: cuando a un fragmento le corresponde un trocito de un téxel de la textura.
- Minimización: cuando a un fragmento le corresponden varios téxeles de la textura.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

6.2.1. Magnificación

Para tratar este problema, WebGL proporciona dos tipos de filtrado:

- Filtro caja: se utiliza el téxel más cercano. Por desgracia, aunque es muy rápido, produce el típico efecto de pixelado que obtenemos al ampliar una imagen (véase figura 6.10).
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);`
- Filtro bilineal: utiliza cuatro téxeles cuyos valores se interpolan linealmente. Este filtro produce un efecto de borrosidad (véase figura 6.11).
  - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);`

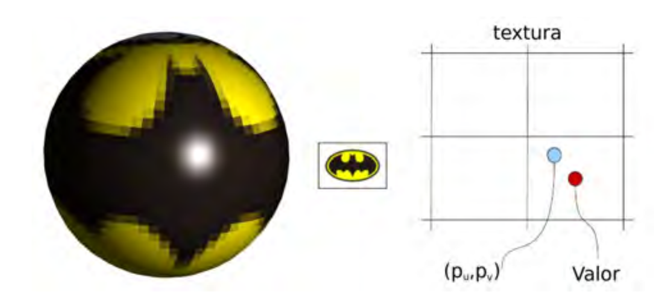


Figura 6.10: Filtro caja de WebGL, devuelve el valor del téxel más cercano y produce el efecto de pixelado

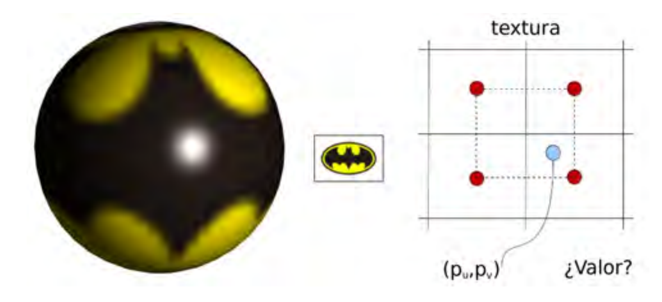


Figura 6.11: Filtro bilineal de WebGL, devuelve la interpolación lineal de cuatro téxeles y produce el efecto de borrosidad

6.2.2. Minimización

WebGL proporciona un método más de filtrado para el problema de la minimización, además del filtro caja y del bilineal comentados en la sección anterior. Se denomina *mipmapping* y consiste en proporcionar, además de la textura original,

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

un conjunto de versiones más pequeñas de la textura, cada una de ellas un cuarto más pequeña que la anterior. A este conjunto de texturas de distintos tamaños se le llama pirámide de texturas.

- Filtro caja y bilineal, respectivamente:
    - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);`
    - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);`
- *Mipmapping*: en tiempo de ejecución, la GPU selecciona la textura cuyo tamaño se acerca más al de la textura en la pantalla (véase figura 6.12). WebGL puede generar la pirámide de texturas de manera automática:
    - `gl.generateMipmap(gl.TEXTURE_2D);`

  Utilizando este método de filtrado, WebGL también puede realizar un filtrado trilineal seleccionando dos texturas, muestreando cada una utilizando un filtro bilineal, e interpolando los dos valores obtenidos:
    - `gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);`

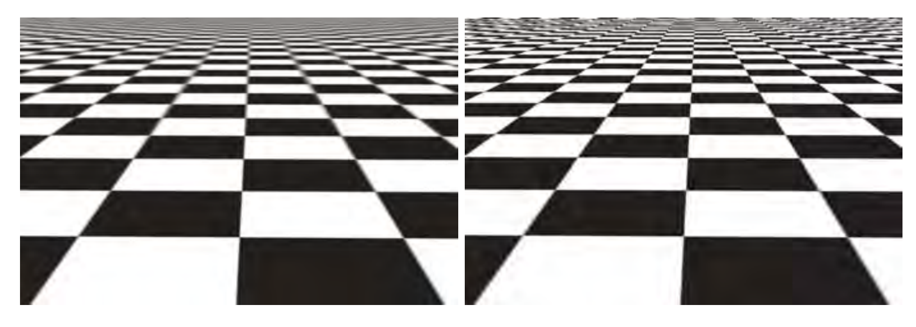


Figura 6.12: Ejemplo de escena con *Mipmapping* (izquierda) y solo filtro bilineal (derecha).

6.3. Texturas en WebGL

Para poder aplicar una textura en WebGL es necesario:

1. crear un objeto textura y establecer todos sus datos, y
2. asignarlo a una unidad de textura.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

En WebGL las texturas se representan mediante objetos con nombre. El nombre no es más que un entero sin signo donde el valor 0 está reservado. Para crear un objeto textura es necesario obtener en primer lugar un nombre que no se esté utilizando. Esta solicitud se realiza con la orden *gl.createTexture*. Se han de solicitar tantos nombres como objetos textura necesitemos, teniendo en cuenta que un objeto textura almacena una única textura.

Una vez creado este objeto, hay que especificar tanto la textura (la imagen 2D en nuestro caso) como los diferentes parámetros de repetición y filtrado. Para especificar la textura se utiliza la siguiente orden:

- `gl.texImage2D (GLenum` *objetivo*`, GLint` *nivel*`, GLenum` *formatoInterno*`, GLsizei` *ancho*`, GLsizei` *alto*`, GLint` *borde*`, GLenum` *formato*`, GLenum` *tipo*`, TexImageSource` *datos*`);`

donde el objetivo será *gl.TEXTURE_2D*. El parámetro *formatoInterno* se utiliza para indicar cuáles de las componentes R, G, B y A se emplean como téxeles de la imagen. Los parámetros *formato*, *tipo* y *datos* especifican, respectivamente, el formato de los datos de la imagen, el tipo de esos datos y una referencia a los datos de la imagen. El parámetro *nivel* se utiliza solo en el caso de usar diferentes resoluciones de la textura, siendo 0 en cualquier otro caso. Y por último, los parámetros *ancho*, *alto* y *borde* se refieren al ancho y alto de la textura, y al tamaño del borde que ha de ser cero. Para que esta orden y las que permiten definir los parámetros de repetición y filtrado tengan efecto sobre la textura, hace falta usar *gl.bindTexture* sobre la textura creada con *gl.createTexture*. El listado 6.3 muestra un ejemplo que incluye la creación de la textura y el establecimiento de sus datos y propiedades.

Listado 6.3: Creación de una textura en WebGL 2.0

```
// Crea un objeto textura
texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);

// Especifica la textura RGB
gl.texImage2D (gl.TEXTURE_2D, 0, gl.RGB, image.width,
    image.height, 0, gl.RGB, gl.UNSIGNED_BYTE, image);

// Repite la textura tanto en s como en t
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);

// Filtrado
gl.texParameteri (gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
    gl.LINEAR_MIPMAP_LINEAR);
gl.texParameteri (gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
    gl.LINEAR);
gl.generateMipmap(gl.TEXTURE_2D);
```


ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Ya solo queda asignar el objeto textura a una unidad de textura. Estas unidades son finitas y su número depende del *hardware*. El consorcio ARB fija que al menos 16 unidades de textura deben existir en WebGL 2.0, pero es posible que nuestro procesador gráfico disponga de más. La orden *gl.activeTexture* especifica el selector de unidad de textura activa. Así, por ejemplo, la orden:

- `gl.activeTexture(gl.TEXTURE3);`

selecciona la unidad de textura 3. A continuación, hay que especificar el objeto textura a utilizar por dicha unidad con la orden *gl.bindTexture*. A cada unidad de textura solo se le puede asignar un objeto textura pero, durante la ejecución, podemos cambiar tantas veces como queramos el objeto textura asignado a una determinada unidad. El listado 6.4 muestra un ejemplo que incluye ambas órdenes.

Listado 6.4: Asignación de un objeto textura a una unidad de textura

```
// Selecciona la unidad de textura 0
gl.activeTexture(gl.TEXTURE0);

// Asocia un objeto textura a la unidad seleccionada
gl.bindTexture (gl.TEXTURE_2D, texture);
```

Una vez creada la textura no hay que olvidar asignar a la variable de tipo *sampler* del *shader* la unidad de textura que ha de utilizar (véase listado 6.5).

Listado 6.5: Asignación de unidad a un *sampler2D*

```
// Obtiene el índice de la variable del shader de tipo sampler2D
program.textureIndex =
  gl.getUniformLocation(program, 'myTexture');

// Indica que el sampler myTexture del shader use
// la unidad de textura 0
gl.uniform1i(program.textureIndex, 0);
```

Como se explicó en el primer punto de este capítulo, los vértices han de proporcionar un nuevo atributo: las coordenadas de textura. En consecuencia, hay que habilitar el atributo (véase listado 6.6) y también especificar cómo se encuentra almacenado (véase listado 6.7).

Listado 6.6: Habilitación del atributo de coordenada de textura

```
// se obtiene la referencia al atributo
program.vertexTexcoordsAttribute =
  gl.getAttribLocation ( program, "VertexTexcoords");

// se habilita el atributo
gl.enableVertexAttribArray(program.vertexTexcoordsAttribute);
```



José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 6.7: Especificación de tres atributos: posición, normal y coordenadas de textura

```
gl.bindBuffer (gl.ARRAY_BUFFER, model.idBufferVertices);
gl.vertexAttribPointer (program.vertexPositionAttribute, 3,
    gl.FLOAT, false, 8*4, 0);
gl.vertexAttribPointer (program.vertexNormalAttribute, 3,
    gl.FLOAT, false, 8*4, 3*4);
gl.vertexAttribPointer (program.vertexTexcoordsAttribute, 2,
    gl.FLOAT, false, 8*4, 6*4);
```

Ejercicios

► **6.9** Ejecuta el programa *c06/aplicaTextura.html*. Experimenta y prueba a cargar diferentes texturas. Como resultado, obtendrás imágenes similares a las de la figura 6.13. Después, edita *aplicaTextura.html* y *aplicaTextura.js* y comprueba cómo se han incorporado todos los cambios necesarios para poder utilizar texturas 2D.



Figura 6.13: Ejemplos obtenidos utilizando texturas en WebGL

6.4. Texturas 3D

Una textura 3D define un valor para cada punto del espacio. Este tipo de texturas resulta perfecta para objetos que son creados a partir de un medio sólido como una talla de madera o un bloque de piedra. Son una extensión directa de las texturas 2D en las que ahora se utilizan tres coordenadas (s, t, r), y donde en lugar de téxeles tenemos vóxeles.

La principal ventaja de este tipo de texturas es que el propio atributo de posición del vértice se puede utilizar como coordenada de textura. Sin embargo, son caras de almacenar y también de filtrar. También son ineficientes cuando se utilizan con modelos de superficies, ya que la mayor parte de la información que la textura almacena nunca se utiliza. Por último, cabe señalar que WebGL soporta este tipo de texturas a partir de la versión 2.0.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

6.5. Mapas de cubo

Un mapa de cubo son seis imágenes cuadradas donde cada una de ellas se asocia a una cara del cubo distinta y que juntas capturan un determinado entorno (véase figura 6.14). Los mapas de cubo se utilizan principalmente para tres aplicaciones:

- *Reflection mapping*
- *Refraction mapping*
- *Skybox*

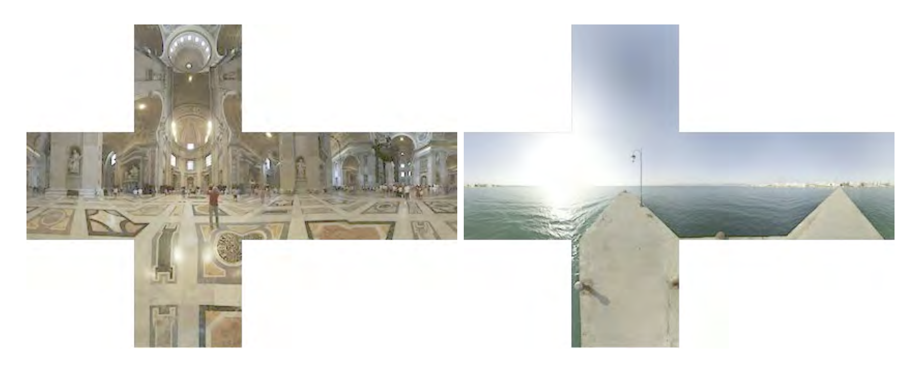


Figura 6.14: Ejemplos de texturas de mapa de cubo (fuente: Emil Persson `http://www.humus.name`)

6.5.1. *Reflection mapping*

Esta técnica de aplicación de textura sirve para simular objetos que reflejan su entorno. En la literatura también aparece con el nombre de *environment mapping*. El objetivo es utilizar una textura que contenga la escena que rodea al objeto y, en tiempo de ejecución, determinar las coordenadas de textura que van a depender del vector dirección del observador o, mejor dicho, de su reflexión en cada punto de la superficie. De esta manera, las coordenadas de textura cambian al moverse el observador, consiguiendo que parezca que el objeto refleja su entorno.

A la textura o conjunto de texturas que se utilizan para almacenar el entorno de un objeto se le denomina mapa de entorno. Este mapa puede estar formado por solo una textura, a la cual se accede utilizando coordenadas esféricas, o por un conjunto de seis texturas cuadradas formando un mapa de cubo. Este último es el caso que se describe en esta sección.

El cálculo de las coordenadas de textura se realiza de la siguiente manera. Para cada punto de la superficie del objeto reflejante se obtiene el vector de reflexión R respecto a la normal N en ese punto de la superficie (véase figura 6.15). Las coordenadas de este vector se van a utilizar para acceder al mapa de cubo y obtener el color. En primer lugar, hay que determinar cuál de las seis texturas se ha de

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

utilizar. Para ello, se elige la coordenada de mayor magnitud. Si es la coordenada x, se utilizan la cara derecha o izquierda del cubo, dependiendo del signo. De igual forma, si es la y se utilizan la de arriba o la de abajo, o la de delante o de detrás si es la z. Después, hay que obtener las coordenadas s y t para acceder a la textura seleccionada. Por ejemplo, si la coordenada R_x del vector de reflexión es la de mayor magnitud, las coordenadas de textura se pueden obtener así:

$$s = \frac{1}{2}\left(\frac{-R_z}{|R_x|} + 1\right) \qquad t = \frac{1}{2}\left(\frac{-R_y}{|R_x|} + 1\right) \tag{6.1}$$

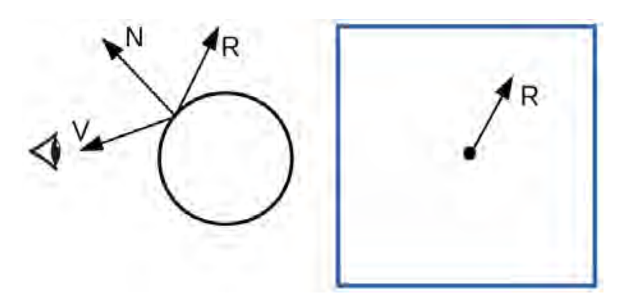


Figura 6.15: Vectores involucrados en *reflection mapping*

En WebGL una textura mapa de cubo se declara de tipo *samplerCube*. El cálculo de las coordenadas de textura se realiza de manera automática al acceder a la textura mediante la función *texture*, que se realiza habitualmente en el *shader* de fragmentos, igual que cuando se accedía a una textura 2D. El vector de reflexión R se calcula para cada vértice mediante la función *reflect*, y el procesador gráfico hará que cada fragmento reciba el vector R convenientemente interpolado. El listado 6.8 muestra el uso de estas funciones en un *shader*. La figura 6.16 muestra un ejemplo del resultado obtenido.



Figura 6.16: El mapa de cubo (que se muestra a la derecha en la figura 6.14) se ha utilizado para simular que el toro está reflejando su entorno

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 6.8: Cambios en el *shader* para *reflection mapping*

```
// Shader de vértices
...
in  vec3 VertexNormal
out vec3 R;  // vector de reflexión en el vértice

void main() {
  ...
  vec4 ecPosition = modelViewMatrix * vec4(vertexPosition,1.0);
  vec3 N          = normalize(normalMatrix * vertexNormal);
  vec3 V          = normalize(vec3(-ecPosition));
  ...
  R = reflect(-V, N);
}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
in vec3 R;    // vector de reflexión interpolado

void main() {
  ...
  fragmentColor = texture(myCubeMapTexture, R);
}
```

6.5.2. *Refraction mapping*

Esta técnica de aplicación de textura se utiliza para simular objetos que la luz atraviesa como hielo, agua, vidrio, diamante, etc. (véase figura 6.17). Las coordenadas de textura se corresponden con el vector de refracción para cada punto de la superficie. El vector de refracción se calcula a partir de los vectores V y N utilizando la ley de Snell. Esta ley establece que la luz se desvía al cambiar de un medio a otro (del aire al agua, por ejemplo) en base a un índice de refracción. El vector de refracción se calcula con la función *refract* para cada vértice en el *shader* de vértices y el procesador gráfico hará que cada fragmento reciba el vector convenientemente interpolado. En el listado 6.9 se muestra el uso de estas funciones en ambos *shaders*. Además, se combina el resultado de la reflexión y la refracción, ya que es habitual que los objetos refractantes también sean reflejantes. El peso se balancea utilizando la variable *refractionFactor*.

6.5.3. *Skybox*

Un *skybox* es un cubo muy grande situado alrededor del observador. El objetivo de este cubo es representar el fondo de una escena que habitualmente contiene elementos muy distantes al observador, como por ejemplo el sol, las montañas, las nubes, etc. Como textura se utiliza un mapa de cubo. Las coordenadas de textura se

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

establecen en el *shader* de vértices, simplemente a partir del atributo de posición de cada vértice del *Skybox*. El procesador gráfico hará que cada fragmento reciba las coordenadas de textura convenientemente interpoladas. En el listado 6.10 se muestran los principales cambios para dibujar el *skybox*.

Listado 6.9: Cambios en el *shader* para *refraction mapping*

```
// Shader de vértices
...
in  vec3 VertexNormal;
out vec3 RefractDir, ReflectDir;

void main() {
  ...
  ReflectDir = reflect(-V, N);
  RefractDir = refract(-V, N, material.refractionIndex);

}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
in vec3 RefractDir, ReflectDir;

void main() {
  ...
  fragmentColor = mix (texture(myCubeMapTexture, RefractDir),
                       texture(myCubeMapTexture, ReflectDir),
                       material.refractionFactor);

}
```



Figura 6.17: Ejemplo de *refraction mapping*

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 6.10: Cambios en el *shader* para *skybox*

```
// Shader de vértices
...
in  vec3 VertexPosition;
out vec3 tcSkybox;  // coordenadas de textura del Skybox

void main() {

  // simplemente asigna a tcSkybox las coordenadas del vértice
  tcSkybox = VertexPosition;
  ...

}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
in vec3 tcSkybox;  // coordenadas de textura interpoladas

void main() {
  ...
  fragmentColor = texture(myCubeMapTexture, tcSkybox);

}
```

Ejercicios

► **6.10** El listado 6.11 contiene el *shader* para que un modelo refleje su entorno y además se pinte el *skybox*. Observa que con el mismo *shader* se pinta tanto el cubo como el modelo. La variable `skybox` se utiliza para saber si se está pintando uno u otro. Por otra parte, la matriz *invT* es la inversa de la matriz modelo-vista del *skybox*. Recuerda que el *skybox* ha de estar centrado en la posición de la cámara. Los archivos *c06/cubeMap.html* y *c06/cubeMap.js* implementan la creación de una textura de mapa de cubo. Edítalos y estúdialos. Observa también las imágenes de la figura 6.18 para ver los resultados que podrías obtener. Interesantes, ¿verdad? Sin duda, un ejemplo inspirado en *Fiat Lux* de Paul Debevec.

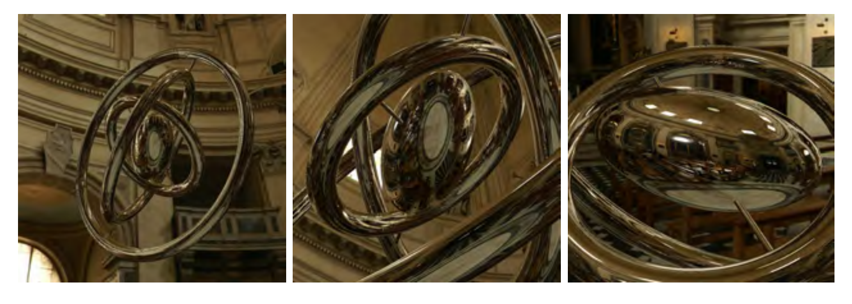


Figura 6.18: Ejemplos obtenidos utilizando *reflection mapping* en WebGL

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 6.11: *Shader* para *skybox* y *reflection mapping*

```
// Shader de vértices
...
uniform bool skybox;
uniform mat3 invT;
out     vec3 R;

void main()  {

  vec3 N          = normalize(normalMatrix * VertexNormal);
  vec4 ecPosition = modelViewMatrix * vec4(VertexPosition,1.0);
  gl_Position     = projectionMatrix * ecPosition;

  if (skybox)
    R = vec3(VertexPosition);
  else
    R = invT * reflect ( normalize( vec3(ecPosition) ), N);

}

// Shader de fragmentos
...
uniform samplerCube myCubeMapTexture;
in vec3 R;

void main() {

  fragmentColor = texture(myCubeMapTexture, R);

}
```

► **6.11** Añade refracción al ejercicio anterior (véase figura 6.19). Utiliza índices de refracción menores que 1 (esto es debido a la implementación de la función *refract* en GLSL). Para cada fragmento, utiliza la función *mix* para que el color final sea un 15 % el valor del reflejo y un 85 % el valor de refracción.




Figura 6.19: Ejemplos obtenidos utilizando *refraction* y *reflection mapping* en WebGL. El índice de refracción es, de izquierda a derecha, de 0,95 y 0,99

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

6.6. *Normal mapping*

Esta técnica consiste en modificar la normal de la superficie para dar la ilusión de rugosidad o simplemente de modificación de la geometría a muy pequeña escala. A esta técnica se le conoce también como *bump mapping*. El cálculo de la variación de la normal se puede realizar en el propio *shader* de manera procedural (véase imagen 6.20), o también se puede precalcular y proporcionar al procesador gráfico como una textura, conocida con el nombre de mapa de normales o *bump map*.

El cálculo de la iluminación se ha de realizar en el espacio de la tangente. Este espacio se construye para cada vértice a partir de su normal, el vector tangente a la superficie en dicho punto y el vector resultante del producto vectorial de esos dos vectores, que se denomina vector binormal. Con los tres vectores se crea la matriz que permite realizar el cambio de base. Como resultado, la normal coincide con el eje Z, la tangente con el eje X y la binormal con el eje Y. Entonces se operan los vectores L y V con esta matriz y hacemos que el procesador gráfico los interpole para cada fragmento. En el *shader* de fragmentos se accede a la textura que almacena el mapa de normales y a partir de esta normal y los vectores L y V interpolados se aplica el modelo de iluminación.

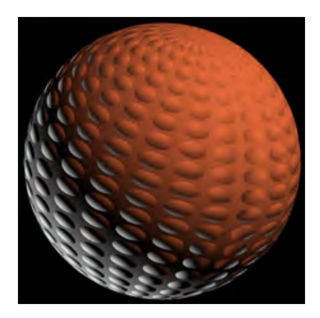
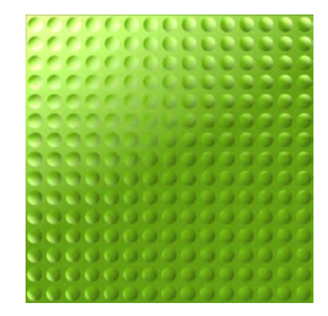



Figura 6.20: Objetos texturados con la técnica de *bump mapping*. La modificación de la normal produce que aparentemente la superficie tenga bultos

6.7. *Displacement mapping*

Esta técnica consiste en aplicar un desplazamiento en cada vértice de la superficie del objeto. El caso más sencillo es aplicar el desplazamiento en la dirección de la normal de la superficie en dicho punto. El desplazamiento se puede almacenar en una textura a la que se le conoce como mapa de desplazamiento. En el *shader* de vértices se accede a la textura que almacena el mapa de desplazamiento y se modifica la posición del vértice, sumándole el resultado de multiplicar el desplazamiento por la normal en el vértice (véase figura 6.21).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151



Figura 6.21: Ejemplo de desplazamiento de la geometría

6.8. *Alpha mapping*

Esta técnica consiste en utilizar una textura para determinar qué partes de un objeto son visibles y qué partes no lo son. Esto permite representar objetos que geométricamente pueden tener cierta complejidad de una manera bastante sencilla a partir de una base geométrica simple y de la textura que hace la función de máscara. Es en el *shader* de fragmentos donde se accede a la textura para tomar esta decisión. Por ejemplo, la figura 6.22 muestra una textura y dos resultados de cómo se ha utilizado sobre el mismo objeto. En el primer caso, se ha utilizado para eliminar fragmentos, produciendo un objeto agujereado cuyo modelado sería bastante más complejo de obtener utilizando métodos tradicionales. En el segundo caso, el valor del *alpha map* se ha utilizado para decidir el color definitivo del fragmento. Con el mismo modelo y cambiando únicamente la textura es muy sencillo obtener acabados muy diferentes (véase figura 6.23).

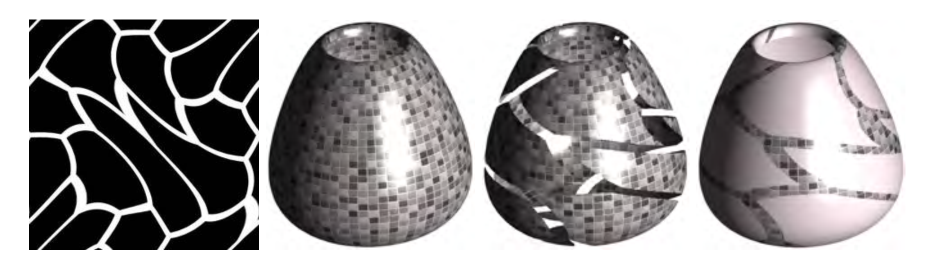


Figura 6.22: Ejemplos de aplicación de la técnica *alpha mapping*

Ejercicios

► **6.12** Con la ayuda de un editor gráfico, crea un *alpha map* o búscalo en internet. Cárgalo como una nueva textura y utilízalo para obtener resultados como los que se muestran en las figuras 6.22 y 6.23.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

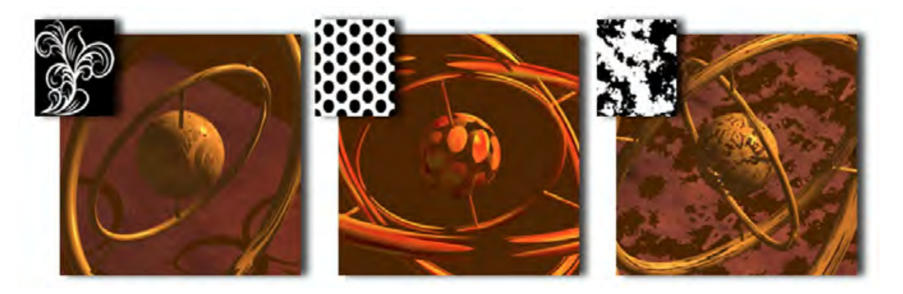


Figura 6.23: Ejemplos de aplicación de diferentes *alpha maps* sobre el mismo objeto

CUESTIONES

▶ **6.1** ¿En qué rango varían las coordenadas de textura en el espacio de la textura? ¿Qué ocurre si una coordenada de textura toma un valor mayor que uno?

▶ **6.2** ¿Cómo se consigue que cada fragmento tenga sus coordenadas de textura cuando estas se especifican por vértice?

▶ **6.3** ¿De qué tipo es la variable que se utiliza para acceder a una unidad de textura?

▶ **6.4** ¿Qué es una unidad de textura? ¿Y un objeto textura?

▶ **6.5** La variable de tipo `sampler2D` contiene un identificador del ¿objeto textura o de la unidad de textura?

▶ **6.6** ¿Dónde se asigna un objeto textura a una unidad de textura, en el *shader* o en el lado de la aplicación?

▶ **6.7** ¿Es posible que desde el *shader* se pueda acceder a varias texturas a la vez? ¿Cómo lo harías?

▶ **6.8** ¿Es posible acceder a una textura desde el *shader* de vértices o solo se puede desde el de fragmentos?

▶ **6.9** ¿Cuántas unidades de textura hay disponibles en el equipo en el que estás haciendo los ejercicios? Recuerda que en el capítulo 1 se muestra un ejemplo de cómo conocer este tipo de información.

▶ **6.10** ¿Qué vectores intervienen en el cálculo de las coordenadas de textura en el método de *Reflection-Mapping*? ¿Y en el de *Refraction-Mapping*?

▶ **6.11** Al dibujar un *Skybox* con WebGL, ¿es necesario que se proporcionen coordenadas de textura como un atributo más de los vértices del cubo?

▶ **6.12** Si deseas utilizar una textura de mapa de cubo para simular el efecto de refracción, ¿cómo proporcionas al *shader* el índice de refracción necesario para obtener el correspondiente vector de refracción, como un atributo del vértice, como variable *uniform*, como variable *in/out* o como una constante?

▶ **6.13** En el método conocido como *Normal Mapping*, indica en qué *shader* (vértices o fragmentos) se accede a la textura que almacena el mapa de normales.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 7

Texturas procedurales

Índice

7.1. Rayado **128**

7.2. Damas **130**

7.3. Enrejado **132**

7.4. Ruido **133**

Una textura procedural se caracteriza porque la información propia de la textura la genera un procedimiento en tiempo de ejecución, es decir, la textura no se encuentra previamente almacenada tal y como se explicó en el capítulo 6. Ahora, existe una función que implementa la textura y a esta función se le llama desde el *shader* para, a partir de unas coordenadas de textura, computar y devolver un valor. La función puede devolver tanto un valor de color como cualquier otro valor que se pueda utilizar para determinar el aspecto final del modelo (véase figura 7.1). Sin duda, el uso de texturas procedurales es una de las grandes virtudes que ofrecen los actuales procesadores gráficos programables.



Figura 7.1: Ejemplo de objeto dibujado con una textura procedural. En este caso, el valor devuelto por la función de textura se utiliza para determinar si hay que eliminar un determinado fragmento

Son varias las ventajas que ofrece el uso de las texturas procedurales. Por ejemplo, una textura procedural, en general, va a consumir menos memoria, ya que al

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

no estar discretizada no presenta los problemas derivados de tener una resolución fija y cualquier aspecto clave de la textura se puede parametrizar para obtener efectos muy diversos utilizando la misma función. Por ejemplo, las copas de la figura 7.1 se han obtenido utilizando exactamente la misma textura procedural, pero para cada una se han asignado valores diferentes a las variables o parámetros que gobiernan el número y tamaño de los agujeros.

También es cierto que no todo son ventajas. Por ejemplo, computacionalmente son más caras y los algoritmos que hay que implementar a veces son muy complejos, incluso también podría ocurrir que los resultados fueran diferentes según el procesador gráfico que se utilice.

En definitiva, combinar texturas basadas en imágenes junto con texturas procedurales será casi siempre lo ideal (véase figura 7.2).

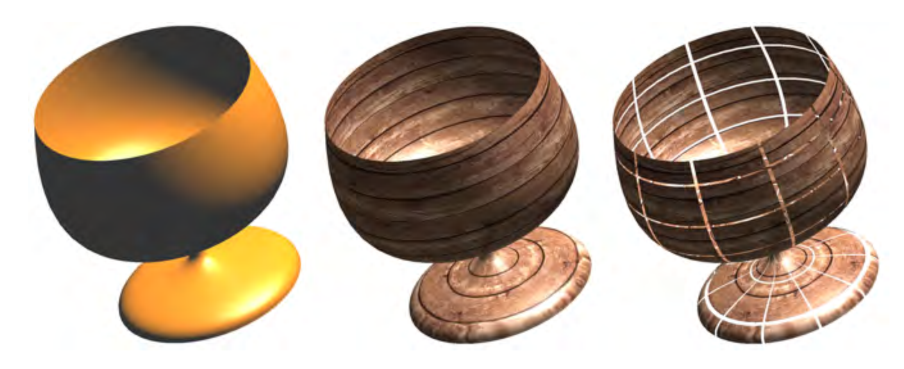


Figura 7.2: Ejemplo de combinación de textura 2D y textura procedural. A la izquierda el objeto es dibujado sin textura, en el centro se le ha aplicado una textura 2D, y a la derecha se ha combinado la textura 2D con una textura procedural

7.1. Rayado

El objetivo de este tipo de textura procedural es mostrar un rayado sobre la superficie del objeto (véase figura 7.3). Se utiliza una coordenada de textura y el rayado que se obtiene siempre será en la dirección de la coordenada elegida. Se aplica un factor de escala sobre la coordenada de textura para establecer el número de rayas. La parte real de la coordenada de textura se compara con una variable que controla el ancho de la raya para saber si el fragmento se ha de colorear con el color de la raya o con el del material del modelo. El listado 7.1 muestra estas operaciones.

Ejercicios

► **7.1** Ejecuta el ejemplo *c07/rayado.html*. Prueba a modificar los parámetros que gobiernan el rayado. Estudia cómo gobiernan el aspecto del rayado y cómo se opera con ellos. ¿Qué coordenada de textura se está interpolando para cada fragmento? No dejes de averiguar el significado de las funciones *fract*, *step* y *mix*.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

▶ 7.2 Modifíca el *shader* de fragmentos en *c07/rayado.html* para que ahora se interpole la otra coordenada de textura (véase figura 7.4).
▶ 7.3 En la figura 7.5 se muestran dos imágenes donde las rayas se obtienen utilizando la función seno. Piensa cómo podrías conseguirlo. Si lo intentas, es muy probable que consigas otros resultados muy interesantes por el camino.

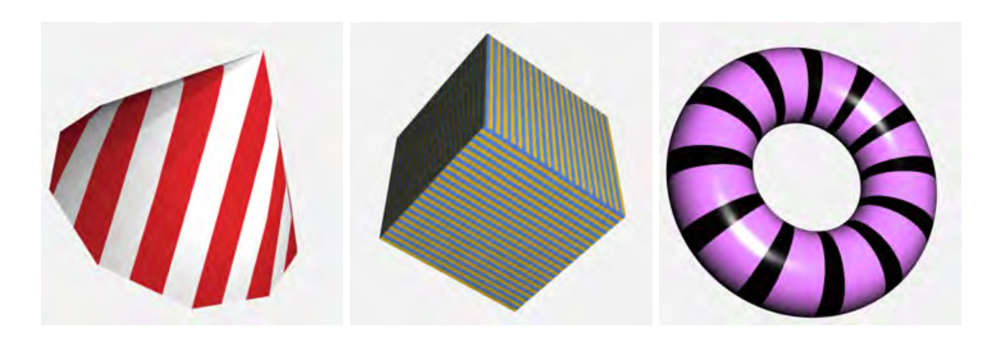


Figura 7.3: Ejemplos del *shader* de rayado

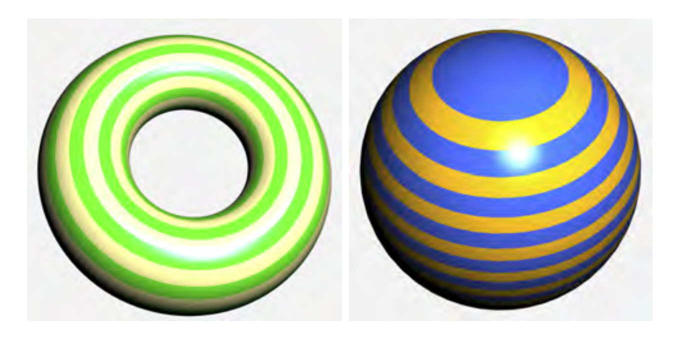


Figura 7.4: Ejemplos del *shader* de rayado utilizando la otra coordenada de textura

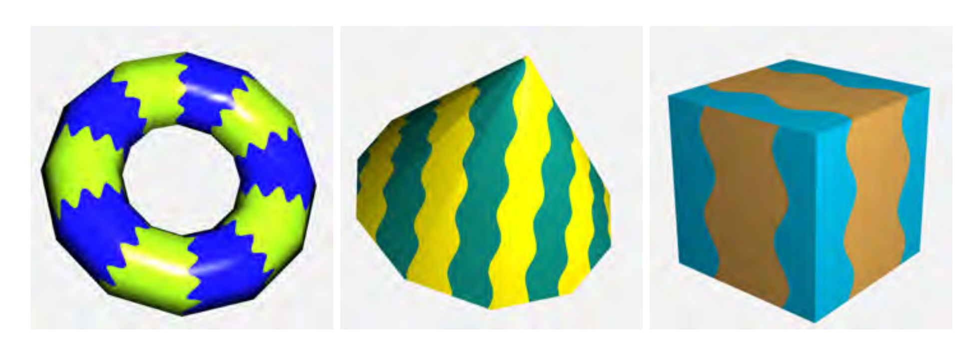


Figura 7.5: Ejemplos del *shader* de rayado utilizando la función seno

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 7.1: *Shader* de rayado

```
// Shader de vértices
...
out float texCoord;

void main()  {
  ...
  texCoord = VertexTexcoords.s;

}
// Shader de fragmentos
...
uniform vec3  StripeColor;    // color de la raya
uniform float Scale;          // número de rayas
uniform float Width;          // ancho de la raya

in   float texCoord;
out  vec4  fragmentColor;

void main() {

  float scaledT = fract(texCoord * Scale);
  float       s = step (Width, scaledT);
  vec3    newKd = mix  (StripeColor, Kd, s);
  ...
  fragmentColor  = vec4 (phong(newKd,N,L,V), 1.0);
}
```

7.2. Damas

Este tipo de textura procedural consiste en presentar la superficie de un objeto como la del tablero del juego de las damas (véase figura 7.6). Se utiliza un factor de escala sobre las coordenadas de textura para establecer el número de cuadrados, en principio el mismo en ambas direcciones. La parte entera de las coordenadas de textura escaladas se utiliza para saber a qué fila y columna pertenece cada fragmento. Sumando ambas y obteniendo el módulo dos se averigua si el resultado es par o impar y, en consecuencia, se sabe si el fragmento pertenece a una casilla blanca o negra. El listado 7.2 recoge los cambios necesarios.

Ejercicios

▶ **7.4** Ejecuta el ejemplo *c7/damas.html*. Prueba a modificar el parámetro que gobierna el número de cuadrados. Ahora, edita el código y modifícalo para que se pueda establecer un factor de escala diferente en cada dirección (véase figura 7.7).

▶ **7.5** En el listado 7.2, ¿con qué finalidad se utiliza la función *floor* en el *shader* de fragmentos?

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

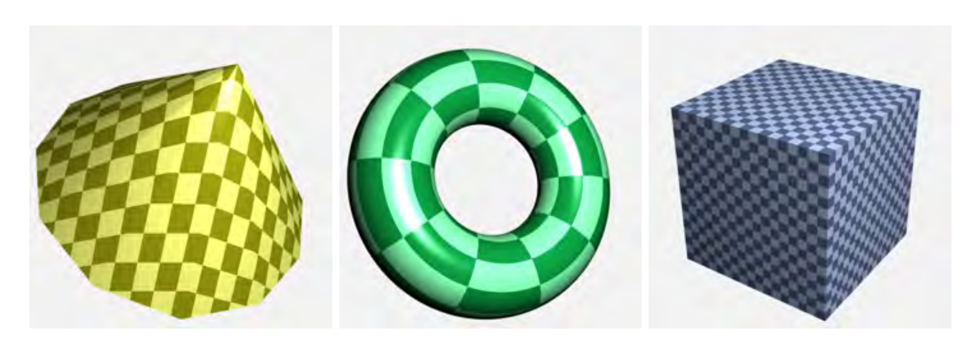


Figura 7.6: Ejemplos del *shader* de damas

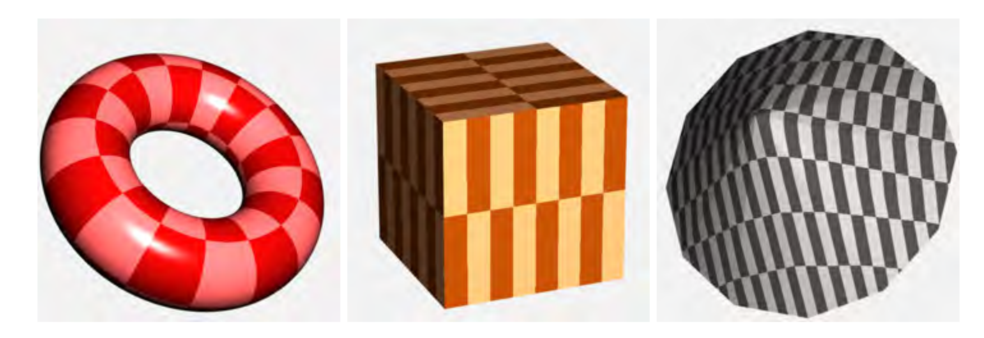


Figura 7.7: Ejemplo del *shader* de damas utilizando un factor de escala distinto para cada dirección

Listado 7.2: *Shader* de damas

```
// Shader de vértices
...
out vec2 texCoord;

void main()  {
  ...
  texCoord = VertexTexcoords;
}

// Shader de fragmentos
...
uniform float Scale;   // número de cuadrados
in      vec2  texCoord;
out     vec4  fragmentColor;

void main() {

  float row   = floor( texCoord.s * Scale );
  float col   = floor( texCoord.t * Scale );
  float res   = mod( row + col , 2.0);
  vec3  newKd = Kd + (res * 0.4);
  ...
  fragmentColor = vec4 (phong(newKd,N,L,V), 1.0);
}
```


ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

7.3. Enrejado

El enrejado consiste en utilizar la orden *discard* para eliminar fragmentos, produciendo en consecuencia agujeros en la superficie del objeto (véanse figuras 7.1 y 7.8). Las coordenadas de textura se escalan para determinar el número de agujeros, utilizando un factor de escala distinto para cada dirección. La parte real se utiliza para controlar el tamaño del agujero. El listado 7.3 recoge los pasos necesarios.

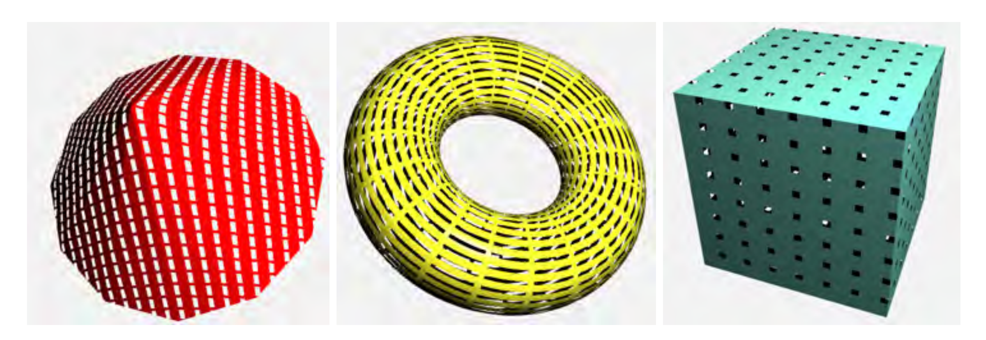


Figura 7.8: Ejemplos del *shader* de enrejado

Listado 7.3: *Shader* de enrejado

```
// Shader de vértices
...
out vec2 TexCoord;

void main()  {
  ...
  TexCoord = VertexTexcoords;
}

// Shader de fragmentos
...
uniform vec2 Scale;
uniform vec2 Threshold;
in      vec2 TexCoord;
out     vec4 fragmentColor;

void main() {

  float ss = fract(TexCoord.s * Scale.s);
  float tt = fract(TexCoord.t * Scale.t);

  if ((ss > Threshold.s) && (tt > Threshold.t))
    discard;
  ...
  fragmentColor = vec4 (phong(N,L,V), 1.0);
}
```


ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Ejercicios

► **7.6** Ejecuta el ejemplo *c07/enrejado.html*. Prueba a modificar los parámetros que gobiernan el número de agujeros y su tamaño.

► **7.7** Ahora, edita el código y modifícalo para que se eliminen los fragmentos que no pertenecen a la superficie de un círculo. Fíjate en los resultados que se muestran en las imágenes de la figura 7.9.



Figura 7.9: Ejemplos de enrejados circulares

► **7.8** De nuevo realiza las modificaciones necesarias para que ahora se eliminen los fragmentos que pertenecen a la superficie de un círculo. Fíjate en el resultado que se muestra en la imágen de la figura 7.10.

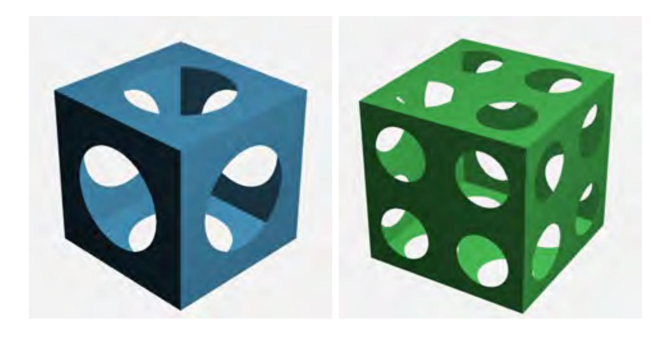


Figura 7.10: Ejemplo de enrejado circular en el que se ha eliminado la superficie del círculo

---

7.4. Ruido

En términos generales, el uso de ruido en informática gráfica resulta muy útil para simular efectos atmosféricos, materiales o simplemente imperfecciones en los objetos. La figura 7.11 muestra algunos ejemplos en los que se ha utilizado una función de ruido como método de textura procedural. A diferencia de otras áreas, la función de ruido que nos interesa ha de ser repetible, es decir, que produzca

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

siempre la misma salida para el mismo valor de entrada y que al mismo tiempo aparente aleatoriedad, es decir, que no muestre patrones regulares.

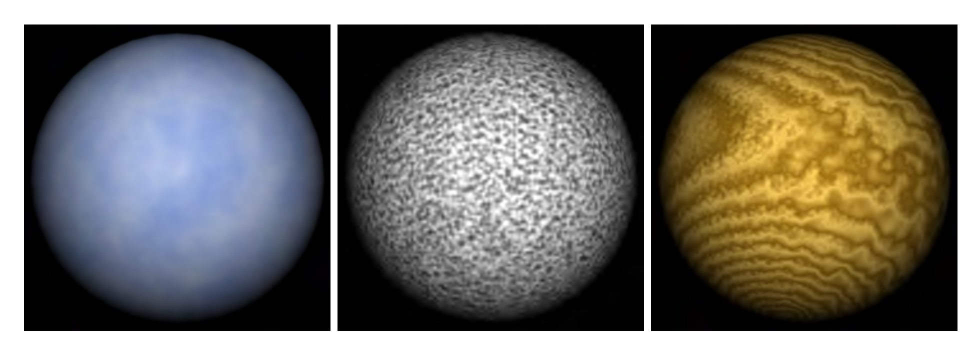


Figura 7.11: Ejemplos obtenidos utilizando una función de ruido como textura procedural

La función de ruido se ha de implementar en el propio *shader*. También se podría implementar en un programa externo, ejecutarlo y almacenar el resultado para proporcionarlo en forma de textura al procesador gráfico, pero en este caso ya no se puede hablar de textura procedural. Por ejemplo, la figura 7.12 muestra diversos ejemplos de objetos en los que se ha utilizado una función de ruido de Perlin en el *shader* de fragmentos para a partir de la posición interpolada de cada fragmento, obtener un valor de ruido y combinarlo de diferentes formas con los valores de material.



Figura 7.12: Ejemplos obtenidos mediante el uso de la función de ruido de Perlin en el *shader* de fragmentos

Ejercicios

▶ **7.9** Ejecuta los ejemplos *c07/nubes.html* y *c07/sol.html* y prueba a modificar los parámetros que gobiernan el aspecto final de los modelos. Después, edita el código y prueba a realizar modificaciones a partir de los valores de ruido obtenidos, seguro que consigues resultados interesantes. Algunos ejemplos de los resultados que se pueden conseguir se muestran en la figura 7.13.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

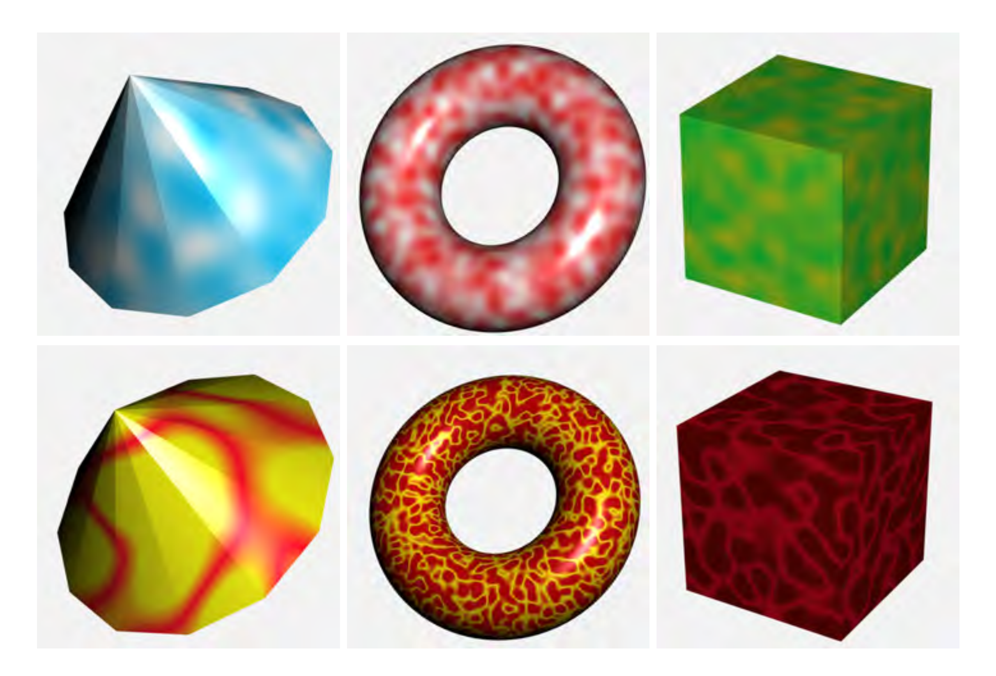


Figura 7.13: Ejemplos obtenidos con la función de ruido de Perlin

CUESTIONES

▶ **7.1** ¿Qué opciones tenemos para utilizar un valor de ruido en WebGL 2.0?

▶ **7.2** ¿Qué desventaja presenta utilizar texturas de ruido frente a utilizar funciones de ruido?

▶ **7.3** Si deseas utilizar texturas procedurales, ¿estás de acuerdo en que nunca es necesario proporcionar coordenas de textura como atributo por cada vértice?

▶ **7.4** Estudia el siguiente *shader* de fragmentos que pinta una superficie como el tablero de las damas. Modifícalo para que el efecto gobernado por la variable *Scale* pueda ser diferente en cada una de las dos direcciones de la textura.

```
uniform float Scale;
in  vec2 TexCoord;
out vec4 fragmentColor;

void main ( ) {

  float row  = floor ( TexCoord.s * Scale );
  float col  = floor ( TexCoord.t * Scale );
  float res  = mod   ( row + col , 2.0 );

  vec3 newKd = Kd + ( res * 0.4 );

  fragmentColor = vec4 ( phong ( newKd ) , 1.0 );

}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **7.5** Estudia el siguiente *shader* de fragmentos que elimina algunos fragmentos produciendo agujeros cuadrados en la superficie de los objetos. Propón las modificaciones oportunas para que en lugar de agujeros cuadrados se observen agujeros circulares y que el radio (válido en el rango $[0, 0.5]$) sea un parámetro que se establezca desde la aplicación.

```
uniform float Scale;
uniform float Threshold;
out vec4 fragmentColor;

void main ( ) {

  float ss = fract ( TexCoord.s * Scale );
  float tt = fract ( TexCoord.t * Scale );

  if ( ( ss < Threshold ) && ( tt < Threshold ) )
    discard;
  fragmentColor = vec4 ( phong ( ) , 1.0 );
}
```

▶ **7.6** Estudia el siguiente *shader* de fragmentos que pinta una superficie como el tablero del popular juego de las damas. Modifícalo para que ambos tipos de casillas se pinten utilizando la misma Kd y que las que inicialmente eran de color más claro sean ahora transparentes con un grado de opacidad del 20 % (y las casillas que inicialmente eran de color más oscuro sigan siendo opacas al 100 %).

```
uniform float Scale;
in  vec2 TexCoord;
out vec4 fragmentColor;

void main ( ) {
  float row   = floor ( TexCoord.s * Scale );
  float col   = floor ( TexCoord.t * Scale );
  float res   = mod (row + col, 2.0);
  vec3  newKd = Kd + ( res * 0.4 );
  fragmentColor = vec4 ( phong ( newKd ) , 1.0 );
}
```

▶ **7.7** Estudia el siguiente *shader* de fragmentos que pinta una superficie a rayas. Modifícalo para que ahora se produzca un cambio gradual entre el color de la raya *StripeColor* y la *Kd*.

```
uniform vec3 StripeColor;
uniform float Scale;
uniform float Width;
in float TexCoord; // coordenada de textura t
out vec4 fragmentColor;

void main ( ) {
  float scaledT = fract ( TexCoord * Scale );
  float s       = step ( Width , scaledT );
  vec3  newKd   = mix ( StripeColor, Kd, s );
  fragmentColor = vec4 ( phong ( newKd ) , 1.0 );
}
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

► **7.8** Estudia el siguiente *shader* de fragmentos y modifícalo para que la raya aparezca en diagonal. Haz también los cambios necesarios para que se pueda establecer el número de rayas deseado.

```
uniform vec3  StripeColor;
in  float TexCoord;    // coordenada de textura s
out vec4  fragmentColor;

void main ( ) {
  float s     = step ( 0.5 , TexCoord );
  vec3  newKd = mix ( StripeColor, Kd, s );
  fragmentColor = vec4 ( phong ( newKd ) , 1.0 );
}
```

► **7.9** Necesitas el modelo de media esfera para incluirlo en tu escena. Sin embargo, solo dispones del modelo de una esfera completa que te proporciona tres atributos por vértice: posición, normal y coordenadas de textura. Desarrolla una textura procedural que como resultado de ordenar el dibujado de la esfera completa produzca que solo se muestre la mitad de ella y que esta mitad siempre se corresponda con el mismo trozo de la esfera (es decir, lo mismo que obtendrías si dispusieras del modelo de media esfera).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 8

Realismo visual

Índice

8.1. Transparencia **139**
8.2. Espejos **143**
8.2.1. Obtener la matriz de simetría 144
8.2.2. Evitar dibujar fuera de los límites 144
8.3. Sombras **147**
8.3.1. Sombras proyectivas 147
8.3.2. *Shadow mapping* 149

Este capítulo presenta tres aspectos básicos en la búsqueda del realismo visual en imágenes sintéticas: transparencia, reflejos y sombras. En la literatura se han presentado numerosos métodos para cada uno de ellos. Al igual que en capítulos anteriores, se van a presentar aquellos métodos que, aun siendo sencillos, consiguen una mejora importante en la calidad visual con poco esfuerzo de programación.

8.1. Transparencia

Los objetos transparentes son muy habituales en el mundo que nos rodea. Suele ocurrir que estos objetos producen un efecto de refracción de la luz, o que la luz cambie alguna de sus propiedades al atravesarlos. Todo esto hace que la inclusión de objetos transparentes en un mundo virtual sea un problema complejo de resolver. En esta sección, se va a abordar el caso más sencillo, es decir, suponer que el objeto transparente es muy fino y no va a producir el efecto de refracción de la luz ni va a modificar las propiedades de las fuentes de luz. Quizá pueda parecer que se simplifica mucho el problema, lo cual es cierto, pero aún así la ganancia visual que se va a obtener es muy alta.

Cuando una escena incluye un objeto transparente, el color de los píxeles cubiertos por dicho objeto depende, además de las propiedades del objeto transparente,

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

de los objetos que hayan detrás de él. Un método sencillo para incluir objetos transparentes en nuestra escena consiste en dibujar primero todos los objetos que sean opacos y dibujar después los objetos transparentes. El grado de transparencia se suministra al procesador gráfico como una cuarta componente en las propiedades de material del objeto, conocida como componente *alfa*. Si *alfa* es 1, el objeto es totalmente opaco, y si es 0 significa que el objeto es totalmente transparente (véase figura 8.1). Así, el color final se calcula a partir del color del *framebuffer* y del color del fragmento de esta manera:

$$C_{final} = alfa \cdot C_{fragmento} + (1 - alfa) \cdot C_{framebuffer} \tag{8.1}$$



Figura 8.1: Tres ejemplos de transparencia con, de izquierda a derecha, *alfa* = 0.3, 0.5 y 0.7

Al dibujar un objeto transparente, el test de profundidad se tiene que realizar de igual manera que al dibujar un objeto opaco y así asegurar que el problema de la visibilidad se resuelve correctamente. Sin embargo, ya que un objeto transparente deja ver a través de él, para cada uno de los fragmentos que supere el test deberá actualizarse el *buffer* de color, pero no el de profundidad, ya que de hacerlo evitaría que otros objetos transparentes situados detrás fuesen visibles. El listado 8.1 recoge la secuencia de órdenes de WebGL necesaria para poder incluir objetos transparentes en la escena. Sin embargo, se ha de tener en cuenta que el navegador compone el resultado de WebGL con la página y que al utilizar el canal *alfa* puede producir un efecto blanquecino en toda la escena. Una forma sencilla de evitarlo es especificar en el fichero *.html* el color negro como color de fondo del canvas:

- `<style> canvas {...; background: black;} </style>`

En el caso de que varios objetos transparentes se solapen en la proyección, el color final en la zona solapada es diferente dependiendo del orden en el que se hayan dibujado (véase figura 8.2). En este caso, habría que dibujar los objetos transparentes de manera ordenada, pintando en primer lugar el más lejano y, en último lugar, el más cercano al observador. Sin embargo, en ocasiones la ordenación no es trivial, como, por ejemplo, cuando un objeto atraviesa otro. En estos

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 8.1: Secuencia de operaciones para dibujar objetos transparentes

```
// dibuja en primer lugar los objetos opacos
...
// activa el cálculo de la transparencia
gl.enable (gl.BLEND);

// especifica la funcion de cálculo de la transparencia
gl.blendFunc (gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);

// impide la actualización del buffer de profundidad
gl.depthMask (false);

// dibuja los objetos transparentes
...
// desactiva el cálculo de la transparencia
gl.disable   (gl.BLEND);

// permite actualizar el buffer de profundidad
gl.depthMask (true);
```

casos, una forma de evitar este problema es establecer la operación de cálculo de la transparencia como un incremento sobre el color acumulado en el *framebuffer*:

$$C_{final} = alfa \cdot C_{fragmento} + C_{framebuffer} \tag{8.2}$$

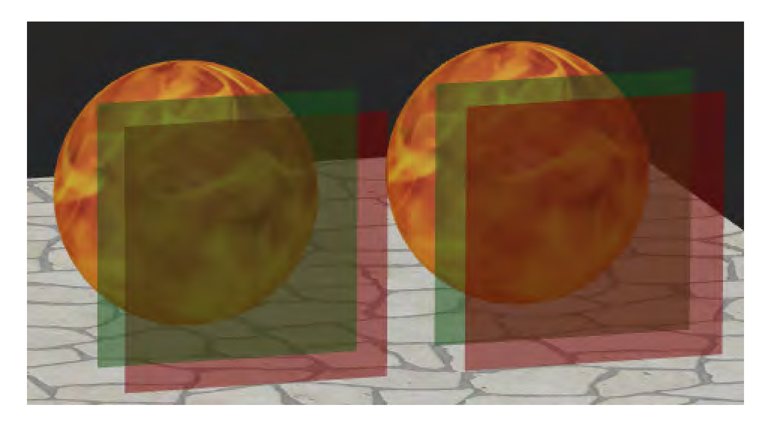


Figura 8.2: Dos resultados diferentes en los que únicamente se ha variado el orden en el dibujado de los objetos transparentes

En WebGL, esto se consigue especificando como función de cálculo *gl.ONE* en lugar de *gl.ONE_MINUS_SRC_ALPHA* (que sería la que corresponde a la ecuación 8.1). De esta manera, el orden en el que se dibujen los objetos transparentes ya no influye en el color final de las zonas solapadas. Por contra, las zonas visibles a través de los objetos transparentes son más brillantes que en el resto, produciendo una diferencia que en general resulta demasiado notable (véase figura 8.3).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

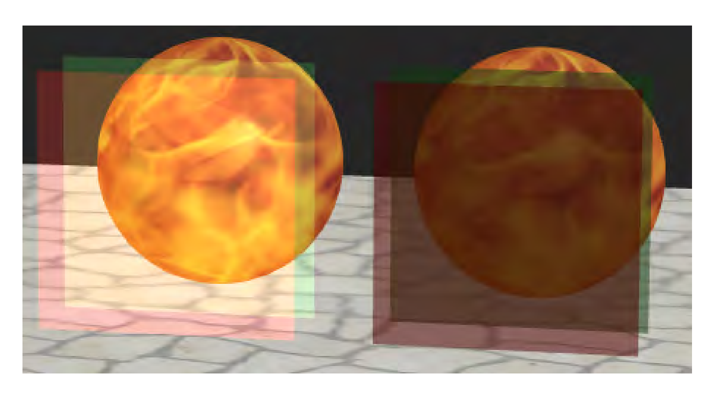


Figura 8.3: Los planos transparentes de la izquierda se han pintado utilizando la función *gl.ONE* mientras que en los de la derecha se ha utilizado *gl.ONE_MINUS_SRC_ALPHA*

Hasta el momento se han utilizado como objetos transparentes polígonos simples, lo que resulta suficiente para simular, por ejemplo, una ventana. Pero, ¿qué ocurre si es un objeto en el que lo que se ve a través de él es a él mismo? En esos casos, se debe prestar atención especial al orden de dibujado. Dado un objeto transparente, una solución muy sencilla consiste en dibujar primero las caras traseras del objeto (que dependen de la posición del observador) y después las caras de delante del objeto. Por suerte, el procesador gráfico implementa en su *pipeline* la eliminación de caras de manera automática. El programador debe habilitarlo (`gl.enable(gl.CULL_FACE)`) e indicar qué caras quiere eliminar, es decir, si quiere eliminar las caras de la parte trasera (`gl.cullFace(gl.BACK)`), o las de la delantera (`gl.cullFace(gl.FRONT)`).

Listado 8.2: Objetos transparentes

```
// Primero pinta los objetos opacos
....
// Después los transparentes
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
gl.enable (gl.BLEND);        // habilita la transparencia
gl.enable (gl.CULL_FACE);    // habilita el face culling
gl.depthMask(false);         // impide actualizar el buffer de prof.

gl.cullFace(gl.FRONT);   // se eliminarán los de cara al observador
drawSolid(exampleCube); // se dibuja el objeto transparente

gl.cullFace(gl.BACK);    // se eliminarán los de la parte trasera
drawSolid(exampleCube); // se vuelve a dibujar el objeto

gl.disable(gl.CULL_FACE);
gl.disable(gl.BLEND);
gl.depthMask(true);
```

El listado 8.2 recoge la secuencia de órdenes de WebGL necesaria para poder visualizar de manera correcta este tipo de objetos transparentes. Además, suele ser conveniente aplicar la iluminación por ambas caras, ya que la parte trasera ahora se

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

ilumina gracias a la transparencia del propio objeto. La figura 8.4 muestra algunos resultados obtenidos con esta técnica.

Ejercicios

▶ **8.1** Observa las dos esferas de la figura 8.2 a las que se les han colocado dos planos transparentes delante de ellas, cada plano de un color distinto, y cuya única diferencia es el orden de dibujado de dichos planos. ¿Cuál de ellas te parece más realista?, ¿por qué?, ¿en qué orden crees que se han pintado los objetos transparentes que aparecen delante de cada esfera?

▶ **8.2** Consulta la información del *pipeline* de WebGL 2.0 y averigua dónde tienen lugar las operaciones de *face culling* y *blending*.

▶ **8.3** Ejecuta el programa *c08/transparencia.html*. Comprueba que en el código figura todo lo necesario para dibujar el objeto de manera transparente. Observa la escena moviendo la cámara y contesta, ¿crees que el objeto transparente se observa de forma correcta desde cualquier punto de vista? Si no es así, ¿por qué crees que ocurre?

▶ **8.4** Edita *c08/transparencia.js*, establece como función de cálculo *gl.ONE* en lugar de *gl.ONE_MINUS_SRC_ALPHA* y observa el nuevo resultado.

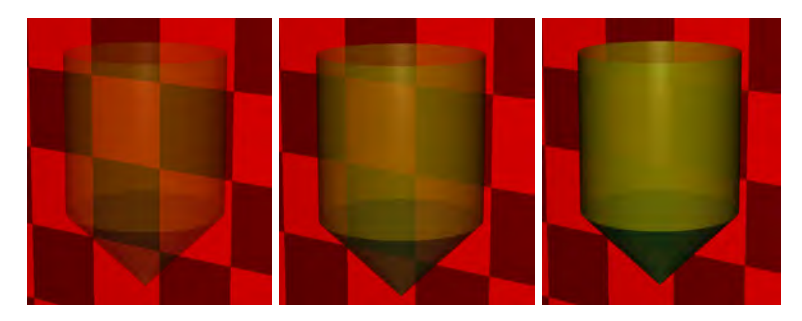


Figura 8.4: Ejemplo de objeto transparente con, de izquierda a derecha, *alfa*= 0.3, 0.5 y 0.7, pintado utilizando el código recogido en el listado 8.2.

8.2. Espejos

Los objetos reflejantes, al igual que los transparentes, son también muy habituales en cualquier escenario. Es fácil encontrar desde espejos puros a objetos que, por sus propiedades de material y también de su proceso de fabricación, como el mármol, por ejemplo, reflejan la luz y actúan como verdaderos espejos. Sin embargo, el cálculo del reflejo es realmente complejo, por lo que en esta sección se muestra un truco muy habitual para conseguir añadir superficies planas reflejantes a nuestra escena (véase figura 8.5) y que de una manera sencilla nos va a permitir obtener muy buenos resultados visuales.

El método consiste en dibujar la escena de forma simétrica respecto al plano que contiene el objeto reflejante (el objeto en el que se vaya a observar el reflejo).

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Hay dos tareas principales: la primera es obtener la transformación de simetría; la segunda es evitar que la escena simétrica se observe fuera de los límites del objeto reflejante.

8.2.1. Obtener la matriz de simetría

Para obtener la matriz de simetría hay que calcular el resultado de la siguiente secuencia de transformaciones: trasladar el plano de simetría al origen, girarlo para hacerlo coincidir con, por ejemplo, el plano $Z = 0$, escalar con un factor de -1 en la dirección Z y, por último, deshacer el giro y la translación anteriores. La matriz resultante M_S es la siguiente, donde P es un punto que pertenece al objeto plano reflejante y N es la normal de dicho plano:

$$M_S = \begin{pmatrix} 1-2N_x^2 & -2N_xN_y & -2N_xN_z & 2(P\cdot N)N_x \\ -2N_xN_y & 1-2N_y^2 & -2N_yN_z & 2(P\cdot N)N_y \\ -2N_xN_z & -2N_yN_z & 1-2N_z^2 & 2(P\cdot N)N_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8.3}$$

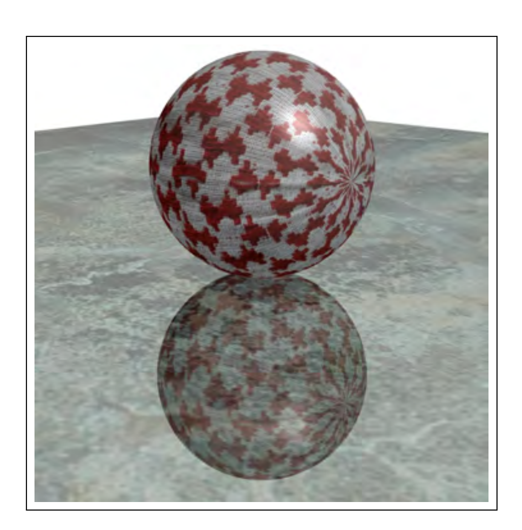


Figura 8.5: Ejemplo de objeto reflejado en una superficie plana

8.2.2. Evitar dibujar fuera de los límites

Para la segunda tarea, la de no dibujar fuera de los límites del objeto reflejante (véanse figuras 8.6 y 8.7), hay varios métodos. Uno de ellos consiste en utilizar el *buffer* de plantilla de la siguiente forma. En primer lugar, se dibuja el objeto reflejante habiendo previamente deshabilitado los *buffers* de color y de profundidad, y también habiendo configurado el *buffer* de plantilla, para que se pongan a 1 los píxeles de dicho *buffer* que correspondan con la proyección del objeto. Después, se habilitan los *buffers* de profundidad y color y se configura el *buffer* de plantilla para rechazar los píxeles que en el *buffer* de plantilla no estén a 1. Entonces, se

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

dibuja la escena simétrica. Después, se deshabilita el *buffer* de plantilla y se dibuja la escena normal. Por último, de manera opcional, hay que dibujar el objeto reflejante utilizando transparencia. El listado 8.3 muestra cómo se realizan estos pasos con WebGL. El ejemplo *c08/espejos.html* recoge todas las operaciones descritas, produciendo imágenes como la de la figura 8.7. Una última observación, en el momento de obtener el contexto es necesario solicitar la creación del *buffer* de plantilla ya que este no se crea por defecto:

- `canvas.getContext(..., {stencil:true});`

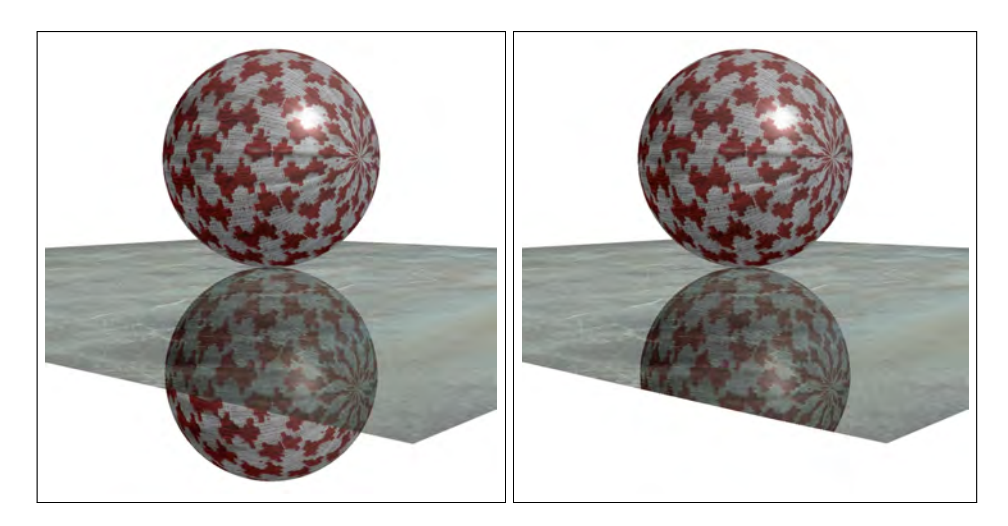


Figura 8.6: Al dibujar la escena simétrica es posible observarla fuera de los límites del objeto reflejante (izquierda). El *buffer* de plantilla se puede utilizar para resolver el problema (derecha)

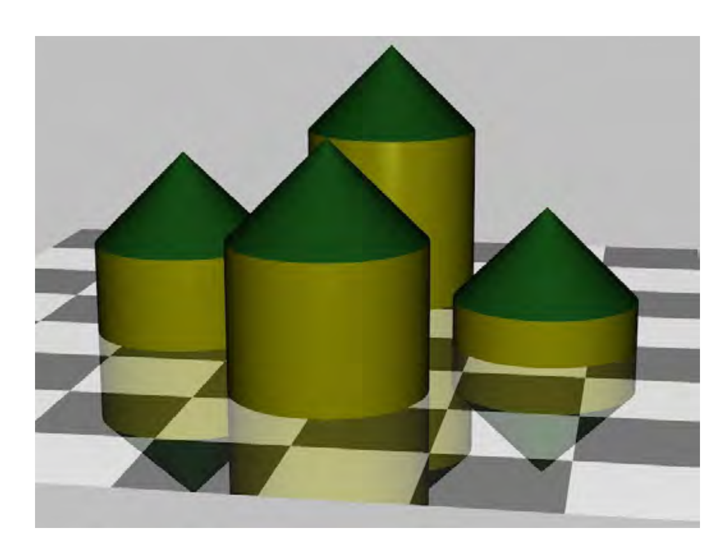


Figura 8.7: Escena en la que el suelo hace de objeto espejo

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 8.3: Secuencia de operaciones para dibujar objetos reflejantes

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT |
     gl.STENCIL_BUFFER_BIT);

// Desactiva los buffers de color y profundidad
gl.disable(gl.DEPTH_TEST);
gl.colorMask(false, false, false, false);

// Establece como valor de referencia el 1
gl.enable(gl.STENCIL_TEST);
gl.stencilOp(gl.REPLACE, gl.REPLACE, gl.REPLACE);
gl.stencilFunc(gl.ALWAYS, 1, 0xFFFFFFFF);

// Dibuja el objeto reflejante
...

// Activa de nuevo los buffers de profundidad y de color
gl.enable(gl.DEPTH_TEST);
gl.colorMask(true, true, true, true);

// Configura el buffer de plantilla
gl.stencilOp(gl.KEEP, gl.KEEP, gl.KEEP);
gl.stencilFunc(gl.EQUAL, 1, 0xFFFFFFFF);

// Dibuja la escena reflejada
...

// Desactiva el test de plantilla
gl.disable(gl.STENCIL_TEST);

// Dibuja la escena normal
...

// Dibuja el objeto reflejante con transparencia
...
```


José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

8.3. Sombras

En el mundo real, si hay fuentes de luz, habrá sombras. Sin embargo, en el mundo de la informática gráfica podemos crear escenarios con fuentes de luz y sin sombras. Por desgracia, la ausencia de sombras en la escena es algo que, además de incidir negativamente en el realismo visual de la imagen sintética, dificulta de manera importante su comprensión, sobre todo en escenarios tridimensionales. Por ejemplo, observa las dos imágenes de la figura 8.8 en la que al añadir la sombra entendemos que el plano está en el aire y no pegado al suelo. Esto ha hecho que en la literatura encontremos numerosos y muy diversos métodos que tratan de aportar soluciones. Por suerte, prácticamente cualquier método que nos permita añadir sombras, por sencillo que sea, puede ser más que suficiente para aumentar el realismo y que el usuario se sienta cómodo al observar el mundo 3D.

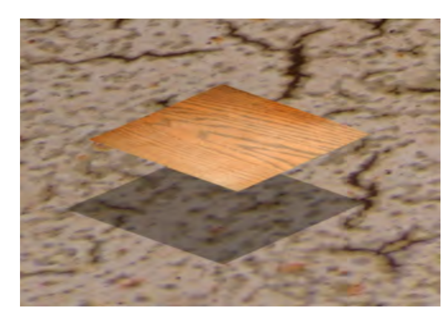



Figura 8.8: En este ejemplo, la misma escena se interpreta de manera distinta dependiendo de si se añade o no la sombra

8.3.1. Sombras proyectivas

Un método simple para el cálculo de sombras sobre superficies planas es el conocido con el nombre de sombras proyectivas. Consiste en obtener la proyección del objeto situando la cámara en el punto de luz y estableciendo como plano de proyección aquel en el que queramos que aparezca su sombra. El objeto sombra, es decir, el resultado de la proyección, se dibuja como un objeto más de la escena, pero sin propiedades de material ni iluminación, simplemente de color oscuro.

Dada una fuente de luz L y un plano de proyección $N \cdot x + d = 0$, la matriz de proyección M es la siguiente:

$$M = \begin{pmatrix} N \cdot L + d - L_x N_x & -L_x N_y & -L_x N_z & -L_x d \\ -L_y N_x & N \cdot L + d - L_y N_y & -L_y N_z & -L_y d \\ -L_z N_x & -L_z N_y & N \cdot L + d - L_z N_z & -L_z d \\ -N_x & -N_y & -N_z & N \cdot L \end{pmatrix} \tag{8.4}$$

Por contra, este método presenta una serie de problemas:

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

- Como el objeto sombra es coplanar con el plano que se ha utilizado para el cálculo de la proyección, habría que añadir un pequeño desplazamiento a uno de ellos para evitar el efecto conocido como *stitching* (véase figura 8.9). WebGL proporciona la orden *gl.polygonOffset* para especificar el desplazamiento, que se sumará al valor de profundidad de cada fragmento siempre y cuando se haya habilitado con *gl.enable (gl.POLYGON_OFFSET_FILL)*.

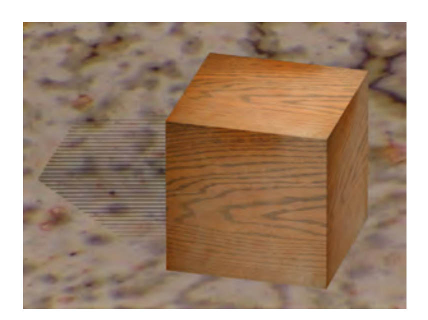


Figura 8.9: Ejemplo de *stiching* producido por ser coplanares el suelo y el objeto sombra

- Hay que controlar que el objeto sombra no vaya más allá de la superficie sobre la que recae (véase figura 8.10). Al igual que en la representación de espejos (véase sección 8.2), el *buffer* de plantilla se puede utilizar para asegurar el correcto dibujado de la escena.

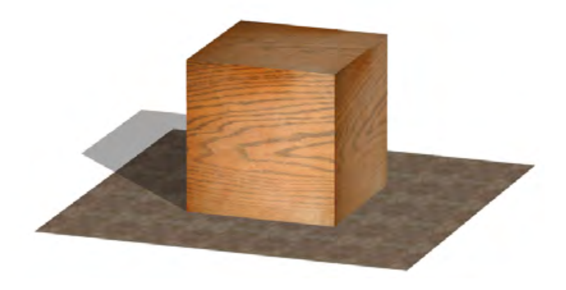


Figura 8.10: El objeto sombra supera los límites de la superficie sobre la que recae

- Las sombras son muy oscuras, pero utilizando transparencia se puede conseguir un resultado mucho más agradable, ya que deja entrever el plano sobre el que se asientan (véase figura 8.11).

- Es muy complejo para superficies curvas o para representar las sombras que caen sobre el propio objeto.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151




Figura 8.11: Ejemplo de sombras proyectivas transparentes

8.3.2. *Shadow mapping*

Si la escena se observa desde la posición donde se ubica la fuente de luz, lo que se consigue es ver justo lo que la fuente de luz ilumina, por lo tanto, se cumple también que estará en sombra lo que la luz no ve. Este método se basa en dibujar primero la escena vista desde la fuente de luz con el objetivo de crear un mapa de profundidad y almacenarlo como textura. Después, se dibuja la escena vista desde la posición del observador pero consultando la textura de profundidad para saber si un fragmento está en sombra y pintarlo de manera acorde.

Para obtener dicho mapa de profundidad con WebGL 2.0, es necesario dibujar la escena contra un *framebuffer object* (FBO). El procesador gráfico puede dibujar en un FBO diferente del creado por defecto, y en ese caso su contenido no es visible al usuario. Este FBO debe tener asociado una textura de profundidad puesto que esa información es la única que nos interesa. Tras crear el FBO, se dibuja la escena y en el *shader* de fragmentos no hay que establecer un color de fragmento (véase imagen de la izquierda en la figura 8.12).

Tras el primer dibujado se consigue que la información de profundidad quede almacenada en la textura de profundidad que se había asociado al FBO. Al dibujar la escena vista desde la posición del observador, cada vértice del modelo se ha de operar también con la matriz de transformación de la cámara situada en la fuente de luz. De esta manera, para cada fragmento se puede comparar su profundidad con la almacenada en la textura y saber si el fragmento está en sombra o no. La figura 8.12 muestra un ejemplo de esta técnica, donde se puede observar el mapa de profundidad obtenido desde la posición de la fuente de luz y la vista con sombras desde la posición de la cámara.

Ejercicios

► **8.5** Ejecuta el programa *c08/sombras.html*. Observa la escena mientras mueves la cámara y contesta, ¿detectas algún tipo de problema en el dibujado de las sombras?, ¿a qué crees que se debe? Modifica el código para que la dimensión de la textura de profundidad sea de un tamaño mucho más pequeño, ¿qué ocurre ahora?, ¿por qué?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

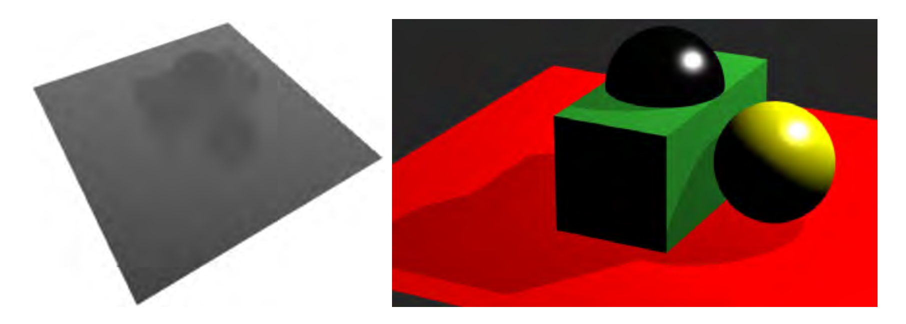


Figura 8.12: Ejemplo de *shadow mapping*. A la izquierda se observa el mapa de profundidad obtenido desde la fuente de luz; a la derecha se muestra la escena con sus sombras

CUESTIONES

▶ **8.1** En el cálculo de la transparencia mediante el operador *over* se utilizan dos valores de color, ¿cuáles son?

▶ **8.2** En WebGL 2.0, ¿cómo se establece la transparencia a nivel de fragmento?

▶ **8.3** ¿Con qué objetivo se utiliza la orden *depthMask* a la hora de trabajar con escenas que incluyen objetos transparentes? Explica también por qué es necesario, es decir, qué tipo de problemas pueden ocurrir si no la utilizas.

▶ **8.4** En el *shader* de fragmentos implementado en el ejemplo *transparencia.html* se realiza el cálculo de la iluminación para ambas caras en el caso de los objetos transparentes, ¿sabrías explicar por qué?

▶ **8.5** Dada una escena que contiene una única fuente de luz y dado un objeto que pertenece a dicha escena, si se hace uso del método de sombras proyectivas ¿cuántas veces se ha de obtener y dibujar el correspondiente objeto sombra?

▶ **8.6** Si quiero implementar el método de sombras proyectivas, ¿por qué y en qué casos necesito utilizar un *buffer* de plantilla?

▶ **8.7** En el método de simulación de espejos mediante simetría, argumenta la necesidad o no de aplicar la matriz de transformación de simetría a las coordenadas, a las normales y a las coordenadas de textura para que el reflejo obtenido sea correcto.

▶ **8.8** En el método de *shadow mapping*, ¿por qué en el primer dibujado no es necesario establecer un color del fragmento? Escribe el código del correspondiente *shader* de fragmentos para el primer dibujado.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 9

Interacción y animación con *shaders*

Índice

9.1. Selección de objetos 151
9.1.1. Utilizando el propio canvas 152
9.1.2. Utilizando un FBO 153
9.2. Animación 154
9.2.1. Eventos de tiempo 155
9.2.2. Encendido / apagado 155
9.2.3. Texturas 156
9.2.4. Desplazamiento 157
9.3. Sistemas de partículas 158

Este capítulo junta técnicas de interacción con métodos básicos de animación con *shaders*. Es un capítulo de índole muy práctica que se acompaña de ejemplos que complementan los métodos explicados.

9.1. Selección de objetos

En una aplicación interactiva es fácil que el usuario pueda señalar objetos de la escena y que, por tanto, la aplicación necesite saber de qué objeto se trata. Habitualmente, el usuario utiliza el ratón para mover el puntero y mediante el botón izquierdo realiza la selección al presionarlo, pero también puede hacerlo con el dedo en el caso de utilizar dispositivos móviles con pantalla táctil. En cualquier caso, como resultado de la interacción se produce un evento que es necesario atender para averiguar las coordenadas del píxel sobre el que se hizo el clic.

Para averiguar las coordenadas del píxel seleccionado hay que tener en cuenta que el origen de coordenadas en el navegador está en la esquina superior izquierda

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

de la página. Por lo tanto, al hacer el clic, el manejador de eventos nos proporciona las coordenadas respecto a dicho origen. Sin embargo, lo que necesitamos conocer son las coordenadas respecto al origen de WebGL que se corresponde con la esquina inferior izquierda del canvas. El listado 9.1 muestra cómo obtener las coordendas correctas para ser utilizadas en WebGL.

Listado 9.1: Conversión de coordenadas para ser utilizadas en WebGL

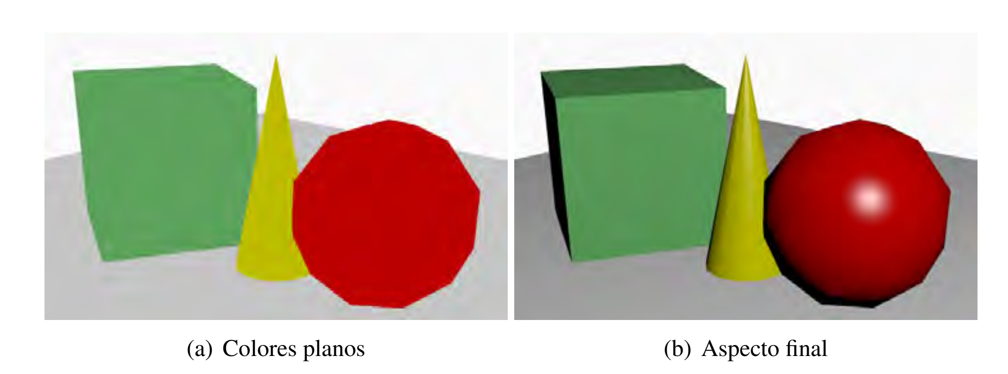
```
rectangle   = event.target.getBoundingClientRect();
x_in_canvas = (event.clientX - rectangle.left);
y_in_canvas = (rectangle.bottom - event.clientY);
```

A partir de aquí hay dos maneras de averiguar de qué objeto se trata, una utilizando solo el canvas y la otra utilizando además un FBO.

9.1.1. Utilizando el propio canvas

Este método consiste en dibujar dos veces la escena de manera consecutiva. En el primer dibujado se pinta cada objeto seleccionable de un color diferente y plano (véase figura 9.1(a)). De esta manera, si tras el primer dibujado se accede al canvas en las coordenadas elegidas por el usuario, se puede averiguar el color del píxel correspondiente, y sabiendo el color se sabe a qué objeto pertenece. Solo quedaría borrar el canvas y pintar de nuevo la escena pero ya con el aspecto final que se ha de mostrar al usuario. Por supuesto, el usuario nunca llega a ver la escena pintada con colores planos, simplemente se dibuja para después acceder al *framebuffer* y conocer el color del píxel.

El listado 9.2 muestra la operación de acceso al color de un píxel que se realiza mediante la función *gl.readPixels*. La variable *pixels* contendrá en consecuencia el valor de color buscado. Después, ya solo resta comparar el valor leído con los utilizados al dibujar los objetos, borrar el *framebuffer* con la orden *gl.clear* y dibujar la escena, esta vez con las técnicas habituales, ya que este nuevo dibujado sí que será el que finalmente termine mostrándose al usuario.



(a) Colores planos

(b) Aspecto final

Figura 9.1: Las dos escenas pintadas para la selección de objetos

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 9.2: Acceso al color de un píxel en el *framebuffer*

```
var pixels = new Uint8Array(4);
gl.readPixels(x_in_canvas, y_in_canvas, 1, 1, gl.RGBA,
              gl.UNSIGNED_BYTE, pixels);
```

Para implementar esta técnica es habitual utilizar un *flag* en el *shader* de fragmentos que indique si el objeto se ha de dibujar o no utilizando un color plano. Este *flag* sería una variable de tipo *uniform bool* que será puesta a verdadero o falso según el caso. Como color plano es buena idea, por ejemplo, aprovechar la propia componente difusa del material del objeto ya que este valor es muy probable que se encuentre disponible en el *shader*. El listado 9.3 muestra un ejemplo muy simple de cómo implementar esta técnica en el *shader* de fragmentos.

Listado 9.3: Uso de un *flag* para decidir si dibujar o no con un color plano

```
fragmentColor = (flag == false) ? vec4 (phong(n,L,V), 1.0)
                                : vec4 (Material.Kd,  1.0);
```

Ejercicios

▶ **9.1** Ejecuta el ejemplo *c09/seleccion.html* que implementa el método de selección descrito en esta sección. Comprueba su funcionamiento. Examina la atención del evento, la obtención de las coordenadas del píxel seleccionado y cómo se determina la primitiva seleccionada en base al color leído. Modifícalo para que el plano del suelo también sea seleccionable.

▶ **9.2** ¿Qué valores de color se obtienen al seleccionar cada uno de los objetos? Comprueba cómo estos valores coinciden con los especificados en las componentes difusas de los respectivos materiales.

9.1.2. Utilizando un FBO

El método anterior dibuja la escena de colores planos para averiguar el objeto seleccionado a partir del momento en que el usuario realiza la selección. Si la escena muestra objetos estáticos, esto no entraña mayor problema. Sin embargo, si los objetos seleccionables están animados, puede ocurrir que la selección del usuario y lo leído en el *framebuffer* no coincida debido al paso del tiempo y la correspondiente actualización de los objetos de la escena. Una manera de evitarlo es tener la escena de colores planos almacenada, de manera que al realizar el clic sea posible realizar la consulta sobre la escena que se está mostrando al usuario y no sobre una nueva.

Esto se puede conseguir utilizando un nuevo objeto *framebuffer* (FBO) para dibujar en él la escena con colores planos. La condición es que este FBO almacene color y profundidad y que tenga el mismo tamaño que el canvas. Cuando el usuario realiza la selección, el FBO ya contiene la imagen dibujada, puesto que en él se ha

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

dibujado la misma escena que el usuario está viendo, solo que con colores planos. De esta manera, lo que se consigue es poder realizar la consulta sobre lo ya dibujado. El listado 9.4 muestra la operación de acceso al color de un píxel. Se puede observar que la única variación es que antes de llamar a la función *gl.readPixels* se activa el FBO creado a próposito y después se vuelve a establecer el *framebuffer* por defecto.

Listado 9.4: Acceso al color de un píxel en el FBO

```
gl.bindFramebuffer(gl.FRAMEBUFFER, myFbo); // selecciona el FBO
var pixels = new Uint8Array(4);
gl.readPixels(x_in_canvas, y_in_canvas, 1, 1, gl.RGBA,
              gl.UNSIGNED_BYTE, pixels);
gl.bindFramebuffer(gl.FRAMEBUFFER, null); // framebuffer normal
```

Ejercicios

► **9.3** Ejecuta el ejemplo *c09/seleccionFbo.html* que implementa el método de selección que utiliza un objeto *framebuffer*.

- Comprueba su funcionamiento.
- Examina cómo se determina la primitiva seleccionada en base al color leído a partir de la escena ya dibujada.
- ¿Hay algún cambio en el *shader* de fragmentos respecto al implementado en el método anterior?
- ¿De qué tamaño es el FBO?, ¿y el canvas?, ¿qué ocurriría si no coincidiesen ambos tamaños?

9.2. Animación

Realizar animación a través de *shaders* puede resultar muy sencillo. Solo necesitamos una variable uniforme en el *shader* y que esta se actualice desde la aplicación con el paso del tiempo. En el *shader* se utilizará dicha variable para modificar cualquiera de las propiedades de los objetos.

Por ejemplo, si se modifica el valor *alfa* que controla la opacidad de un objeto, podemos conseguir que este aparezca o se desvanezca de forma gradual; también modificar parámetros de las fuentes de luz haciendo que, por ejemplo, la intensidad de la luz aumente o decaiga de forma gradual o que los objetos cambien su color. Por supuesto, también podemos modificar la matriz de transformación del modelo cambiando la posición, el tamaño o la orientación de algún objeto de la escena o, por qué no, añadir dos materiales a un objeto y hacer que un objeto cambie de material con una simple interpolación lineal.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

9.2.1. Eventos de tiempo

JAVASCRIPT proporciona una orden para especificar que una determinada función sea ejecutada transcurrido un cierto tiempo. La disponibilidad de una función de este tipo es fundamental para actualizar la variable del *shader* que se utiliza para generar la animación. La función es la siguiente:

- `myVar = setTimeout(updateStuff, 40);`

El valor numérico indica el número de milisegundos que han de transcurrir para que se llame a la función *updateStuff*. Una vez transcurrido dicho tiempo, esa función se ejecutará lo antes posible. Si se desea que la función se ejecute otra vez al cabo de un nuevo periodo de tiempo, ella misma puede establecerlo llamando a la función *setTimeout* antes de finalizar. Otra alternativa es utilizar la siguiente orden, que produce la ejecución de *updateStuff* cada 40 ms:

- `myVar = setInterval(updateStuff, 40);`

Y para suspender la ejecución:

- `clearInterval(myVar);`

9.2.2. Encendido / apagado

Un efecto muy simple de animación consiste en que una propiedad tome dos valores diferentes que van alternándose a lo largo del tiempo, como podrían ser la simulación del parpadeo de una luz al encenderse o el mal funcionamiento de un tubo fluorescente. Para implementar esta técnica es habitual utilizar un *flag* en el *shader* de fragmentos que indique si el objeto se ha de dibujar de una manera u otra. Este *flag* sería una variable de tipo *uniform bool* que será puesta a verdadero o falso según el caso. Por ejemplo, es buena idea aprovechar la propia componente ambiente del material del objeto para el estado de apagado ya que este valor es muy probable que se encuentre disponible en el *shader*. El listado 9.5 muestra un ejemplo muy simple de cómo implementar esta técnica en el *shader* de fragmentos en el que la variable *apagado* indica el estado, es decir, si encendido o apagado).

Listado 9.5: *Shader* para encendido / apagado

```
// Shader de fragmentos
...
uniform bool apagado; // almacena el estado
...
void main() {
  ...
  fragmentColor = (apagado == false) ? vec4 (phong(n,L,V), 1.0):
                                       vec4 (Material.Ka,   1.0);
}
```


ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

En la aplicación es habitual que dicha variable esté gobernada por un simple proceso aleatorio. Por ejemplo, la función *setFlickering* en el listado 9.6 recibe un valor como parámetro que se compara con un número aleatorio. De esta manera, se puede controlar la preferencia hacia uno de los dos estados posibles.

Listado 9.6: Función que controla el encendido / apagado

```
function setFlickering(value) {

  if (Math.random()> value)  // Math.random en [0..1]
    gl.uniform1i (program.apagadoIndex, false);
  else
    gl.uniform1i (program.apagadoIndex, true);

}
```

Ejercicios

► **9.4** Ejecuta el ejemplo *c09/onOff.html* que implementa el método de animación de encendido / apagado. Comprueba su funcionamiento. Examina cómo se establece un valor diferente de parpadeo para cada primitiva, de manera que algunas se ven más tiempo encendidas y otras más tiempo apagadas. Realiza las modificaciones necesarias para que, en lugar de encendido / apagado, sea encendido / sobreiluminado.

9.2.3. Texturas

Modificar las coordenadas de textura con el tiempo es algo sencillo y de lo que se pueden obtener resultados muy interesantes. Por ejemplo, en el caso de ser una textura 2D, se podría simular un panel publicitario rotativo. En el caso de utilizar una textura procedural, por ejemplo, el *shader* de nubes utilizado en el ejemplo *c07/nubes.html*, modificar las coordenadas de textura permite que su aspecto cambie suavemente con el tiempo. En cualquier caso, solo es necesario utilizar una variable que se incremente con el paso del tiempo (véase listado 9.7) y que a su vez se utilice para incrementar las coordenadas de textura en el *shader*, como se muestra en el listado 9.8. De esta manera, a cada instante de tiempo las coordenadas de textura de cada vértice son diferentes, produciéndose el efecto de animación.

Ejercicios

► **9.5** Ejecuta el ejemplo *c09/nubes.html* que implementa el método de animación basado en modificar las coordenadas de textura con el tiempo. Comprueba su funcionamiento. Examina cómo se modifican las coordenadas de textura para acceder a la función de ruido. Ahora ponlo en práctica. Parte del ejemplo *c06/aplicaTexturas.html* y modifícalo para que la textura se desplace sobre la superficie de las primitivas con el paso del tiempo.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 9.7: Función que actualiza el desplazamiento de la textura con el tiempo

```
var texCoordsOffset = 0.0, Velocity = 0.01;

function updateTexCoordsOffset() {

  texCoordsOffset += Velocity;
  gl.uniform1f(program.texCoordsOffsetIndex, texCoordsOffset);

  requestAnimationFrame(drawScene);

}

function initWebGL() {
  ...
  setInterval(updateTexCoordsOffset,40);
  ...
}
```

Listado 9.8: *Shader* para actualizar las coordenadas de textura con el tiempo

```
// Shader de fragmentos
...
uniform float texCoordsOffset;
...
void main() {
  ...
  vec2 newTexCoords = texCoords;
  newTexCoords.s    += texCoordsOffset;
  fragmentColor      = texture(myTexture, newTexCoords);
  ...
}
```

9.2.4. Desplazamiento

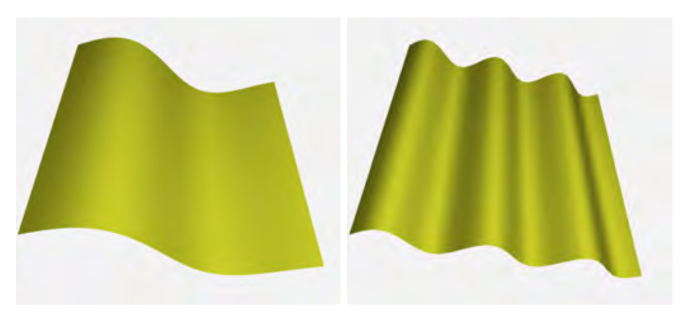
En la sección 6.7 se explicó el método que permite utilizar una textura como mapa de desplazamiento para que en tiempo de ejecución cada vértice se desplace a partir del valor leído del mapa. Ahora, lo que se persigue es animar la geometría, haciendo que el desplazamiento de cada vértice no sea siempre el mismo, sino que cambie con el tiempo. La figura 9.2(a) muestra dos ejemplos en los que en el *shader* de vértices se ha utilizado la función seno con diferente amplitud de onda para calcular el desplazamiento en función del tiempo transcurrido. En la figura 9.2(b) se muestra el resultado de utilizar una función de ruido para determinar el desplazamiento. El valor de tiempo se utiliza como parámetro de entrada de la función de ruido, obteniéndose el efecto de una bandera al viento.

Ejercicios

► **9.6** El ejemplo *c09/vd.html* implementa el método de animación basado en desplazamiento. Ejecútalo y, por ejemplo, prueba a escalar el valor del desplazamiento.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151



(a) Desplazamiento producido por la función seno



(b) Desplazamiento producido por la función de ruido

Figura 9.2: Objetos animados con la técnica de desplazamiento

9.3. Sistemas de partículas

Los sistemas de partículas son una técnica de modelado para objetos que no tienen una frontera bien definida como, por ejemplo, humo, fuego o un espray. Estos objetos son dinámicos y se representan mediante una nube de partículas que, en lugar de definir una superficie, definen un volumen. Cada partícula nace, vive y muere de manera independiente. En su periodo de vida, una partícula cambia de posición y de aspecto. Atributos como posición, color, transparencia, velocidad, tamaño, forma o tiempo de vida se utilizan para definir una partícula. Durante la ejecución de un sistema de partículas, cada una de ellas se debe actualizar a partir de sus atributos y de un valor de tiempo global.

Las figuras 9.3 y 9.4 muestran ejemplos de un sistema en el que cada partícula es un cuadrado y pretenden modelar respectivamente un mosaico y pequeñas banderas en un escenario deportivo. En ambos casos, a cada partícula se le asigna un instante de nacimiento aleatorio de manera que las piezas del mosaico, o las banderas, aparecen en instantes de tiempo diferentes. Mientras están vivas, para cada partícula se accede a una textura de ruido utilizando la variable que representa el paso del tiempo y así no acceder siempre al mismo valor de la textura con el fin de actualizar su posición. A cada partícula también se le asigna de forma aleatoria un valor de tiempo final que representa el instante en que la partícula debe desaparecer.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

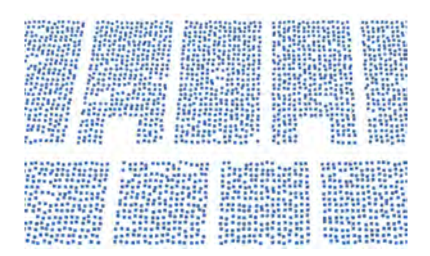
 

Figura 9.3: Animación de un mosaico implementado como sistema de partículas



Figura 9.4: Animación de banderas implementada como sistema de partículas

Todas las partículas, junto con sus atributos, pueden ser almacenadas en un *buffer object*. De esta manera, en el *shader* de vértices se determina el estado de la partícula, es decir, si aún no ha nacido, si está viva o si por el contrario ya ha muerto. En el caso de estar viva, es en dicho *shader* donde se implementa su comportamiento. Por lo tanto, la visualización de un sistema de partículas se realiza totalmente en el procesador gráfico sin carga alguna para la CPU. Para simplificar el dibujado de un sistema de partículas se asume que las partículas no colisionan entre sí, no reflejan luz y no producen sombras sobre otras partículas.

Un ejemplo de creación de un sistema de partículas se muestra en el listado 9.9. En concreto se crean diez mil partículas; a cada partícula se le asigna una velocidad y una posición a lo largo del eje X, ambas aleatorias, y un valor de nacimiento. Esta información se almacena en el vector *particlesInfo*, el cual se transfiere a un *buffer object*.

El listado 9.10 muestra cómo se ordena el dibujado del sistema de partículas. En este ejemplo, cada partícula consta de dos atributos, posición e instante de nacimiento, y el sistema se dibuja como una colección de puntos.

Por útltimo, en el *shader* de vértices se comprueba si la partícula ha nacido y, si es así, se calcula su posición a partir del valor de posición X y el valor de velocidad almacenado en la posición Y, junto con el valor del tiempo transcurrido. El listado 9.11 muestra este último paso. La figura 9.5 muestra dos ejemplos en los que únicamente cambia el tamaño del punto.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 9.9: Cortina de partículas

```
var numParticles    = 10000;

function initParticleSystem() {

  var particlesInfo = [];

  for (var i= 0; i < numParticles; i++) {

    // velocidad
    var alpha    = Math.random();
    var velocity = (0.1 * alpha) + (0.5 * (1.0 - alpha));

    // posición
    var x = Math.random();
    var y = velocity;
    var z = 0.0;

    particlesInfo[i * 4 + 0] = x;
    particlesInfo[i * 4 + 1] = y;
    particlesInfo[i * 4 + 2] = z;
    particlesInfo[i * 4 + 3] = i * 0.00075; // nacimiento
  }

  program.idBufferVertices = gl.createBuffer ();
  gl.bindBuffer (gl.ARRAY_BUFFER, program.idBufferVertices);
  gl.bufferData (gl.ARRAY_BUFFER,
                 new Float32Array(particlesData),
                 gl.STATIC_DRAW);
}
```

Listado 9.10: Dibujado del sistema de partículas

```
function drawParticleSystem() {

  gl.bindBuffer (gl.ARRAY_BUFFER, program.idBufferVertices);
  gl.vertexAttribPointer (program.vertexPositionAttribute,
                          3, gl.FLOAT, false, 4*4,   0);
  gl.vertexAttribPointer (program.vertexStartAttribute,
                          1, gl.FLOAT, false, 4*4,   3*4);

  gl.drawArrays (gl.POINTS, 0, numParticles);

}
```

Ejercicios

▶ **9.7** El ejemplo *c09/particulas.html* implementa la animación de un sistema de partículas. Ejecútalo y experimenta con él para comprenderlo mejor.
▶ **9.8** Modifica el ejemplo para que el sistema de partículas tenga profundidad. Tendrás que modificar tanto el *js* como el *shader*. El efecto que se consigue es muy interesante.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 9.11: *Shader* de vértices para el sistema de partículas

```
...
in vec3  VertexPosition;
in float VertexStart;

void main()  {

  vec3 pos = vec3(0.0);

  if (Time > VertexStart) {        // si ha nacido
    float t = Time - VertexStart;
    if (t < 2.0) {                 // si aún vive
      pos.x = VertexPosition.x;
      pos.y = VertexPosition.y * t;
      alpha = 1.0 - t / 2.0;
    }
  }

  vec4 ecPosition = modelViewMatrix * vec4(pos,1.0);

  gl_Position    = projectionMatrix * ecPosition;
  gl_PointSize   = 6.0;

}
```

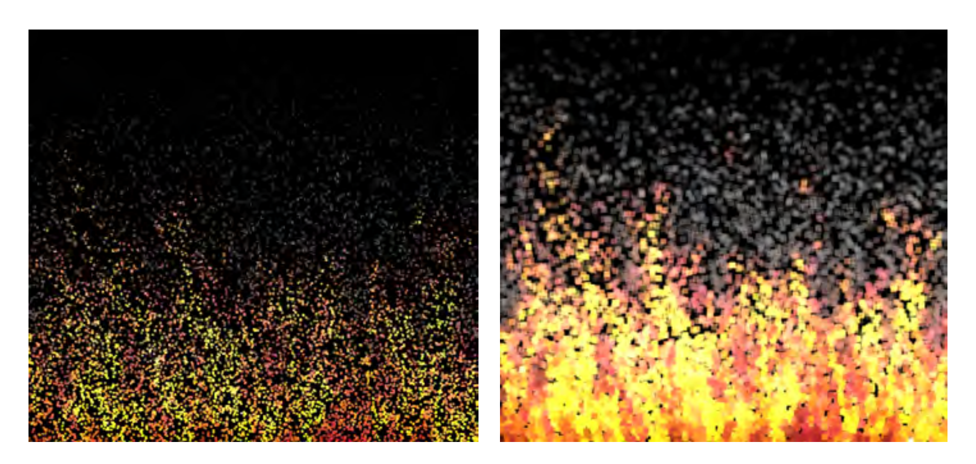


Figura 9.5: Ejemplo de sistema de partículas dibujado con tamaños de punto diferentes

CUESTIONES

▶ **9.1** Justo después de que un usuario realice la acción de selección de un objeto es necesario averiguar de qué objeto se trata, ¿cuál de los dos métodos vistos en este capítulo no necesita dibujar la escena justo a continuación del *clic* para averiguar de qué objeto se trata?

▶ **9.2** De entre los dos métodos vistos en este capítulo para selección de objetos, ¿cuál es la principal ventaja que proporciona el método que usa un FBO y en qué casos se produce?

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

► **9.3** En el proceso de selección de un objeto, ¿por qué no puedo utilizar directamente las coordenadas obtenidas a través de la variable $event$ con la orden *gl.readPixels*?

► **9.4** En un sistema de partículas genérico, ¿cómo se proporciona al *shader* el valor de nacimiento de cada partícula?

► **9.5** Supón que en tu escenario virtual deseas incluir una valla publicitaria. Esta valla ha de mostrar dos imágenes alternas en el tiempo, intercambiándolas cada 10 segundos. El cambio de una imagen a otra se realiza mediante un desplazamiento vertical y dura un segundo. Detalla cómo lo harías, tanto en la parte de *javascript* como en el *shader*. Trata de ser lo más especifico posible (cómo modelarías la valla, qué variables necesitas para la animación, cómo controlarías el tiempo, etc).

► **9.6** Estás modelando un planeta y quieres animarlo haciendo que gire sobre sí mismo. Quizá lo más sencillo sea utilizar una transformación de rotación, de manera que el ángulo de giro se incremente con el tiempo. Propón una alternativa a utilizar transformaciones que consiga el mismo efecto visual (de que el planeta gira alrededor de si mismo), detalla la respuesta todo lo que puedas como si un tercero tuviese que implementarlo a partir de tu explicación.

ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Capítulo 10

Procesamiento de imagen

Índice

10.1. Apariencia visual **163**
10.1.1. *Antialiasing* 163
10.1.2. Corrección gamma 166
10.2. Posproceso de imagen **167**
10.2.1. Brillo 167
10.2.2. Contraste 168
10.2.3. Saturación 168
10.2.4. Negativo 168
10.2.5. Escala de grises 170
10.2.6. Convolución 170
10.3. Transformaciones **171**

Desde sus orígenes, OpenGL ha tenido en cuenta en el diseño de su *pipeline* la posibilidad de manipular imágenes sin asociarle geometría alguna. Sin embargo, no es hasta que se produce la aparición de los procesadores gráficos programables cuando de verdad se puede utilizar OpenGL como una herramienta para procesado de imágenes, consiguiendo aumentar de manera drástica la capacidad de analizar y modificar imágenes, así como de generar una amplia variedad de efectos (véase figura 10.1).

10.1. Apariencia visual

10.1.1. *Antialiasing*

Se conoce como esfecto escalera o dientes de sierra, o más comúnmente por su término en inglés *aliasing*, al artefacto gráfico derivado de la conversión de entidades continuas a discretas. Por ejemplo, al visualizar un segmento de línea se

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151



Figura 10.1: Ejemplo de procesado de imagen. A la imagen de la izquierda se le ha aplicado un efecto de remolino, generando la imagen de la derecha

convierte a una secuencia de píxeles coloreados en el *framebuffer*, siendo claramente perceptible el problema, excepto si la línea es horizontal o vertical (véase figura 10.2). Este problema es todavía más fácil de percibir, y también mucho más molesto, si los objetos están en movimiento.



Figura 10.2: En la imagen de la izquierda se observa claramente el efecto escalera, que se hace más suave en la imagen de la derecha

Nos referimos con *antialiasing* a las técnicas destinadas a eliminar ese efecto escalera. Hoy en día, la potencia de los procesadores gráficos permite que desde el propio panel de control del controlador gráfico el usuario pueda solicitar la solución de este problema e incluso establecer el grado de calidad. Hay que tener en cuenta que, a mayor calidad del resultado, mayor coste para la GPU, pudiendo llegar a producir cierta ralentización en la interacción con nuestro entorno gráfico. Por este motivo, las aplicaciones gráficas exigentes con el *hardware* gráfico suelen ofrecer al usuario la posibilidad de activarlo como una opción.

Supersampling

El método de *supersampling* se basa en tomar más muestras por cada píxel. De esta manera, el valor final de un píxel p se obtiene como resultado de la combinación de todas sus muestras. Hay que definir un patrón de muestreo y también se puede asignar un peso diferente a cada muestra.

$$p(x, y) = \sum_{i=1}^{n} w_i c(i, x, y) \tag{10.1}$$

La figura 10.3 muestra un ejemplo del funcionamiento del método donde, en lugar de utilizar una única muestra, se utilizan cuatro. En ese ejemplo, dos de las muestras quedan cubiertas por la proyección de la primitiva gráfica y el color final del píxel es el valor medio ponderado de los valores obtenidos para las cuatro muestras realizadas.

ISBN: 978-84-17429-87-4

DOI: http://dx.doi.org/10.6035/Sapientia151

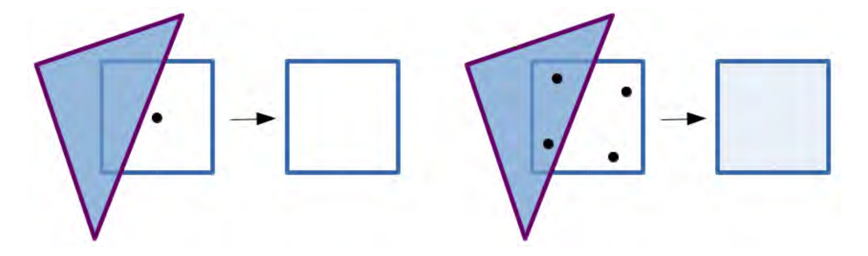


Figura 10.3: Ejemplo de funcionamiento del *supersampling*

La implementación más popular de este método se conoce con el nombre de *full scene anti-aliasing*, FSAA. Al utilizar esta técnica es necesario disponer de un *framebuffer* cuyo tamaño sea n veces mayor, donde n es el número de muestras, ya no solo para almacenar el color, sino también, por ejemplo, para guardar la profundidad de cada muestra. Este método procesa cada muestra de manera independiente, por lo que es bastante costoso, dado que el número de píxeles se multiplica fácilmente por cuatro, ocho o incluso dieciséis.

Multisampling

El método conocido por *multi-sampling anti-aliasing*, MSAA, se basa en muestrear cada píxel n veces para averiguar el porcentaje del píxel cubierto por la primitiva. El *shader* de fragmentos solo se ejecuta una vez por fragmento, a diferencia del método *full scene anti-aliasing*, donde dicho *shader* se ejecutaba para cada muestra. WebGL 2.0 implementa este método (véase figura 10.4) y por defecto está habilitado. En el caso de no quererlo hay que desactivarlo de manera específica al obtener el contexto:

- `canvas.getContext("webgl2",{antialias:false});`

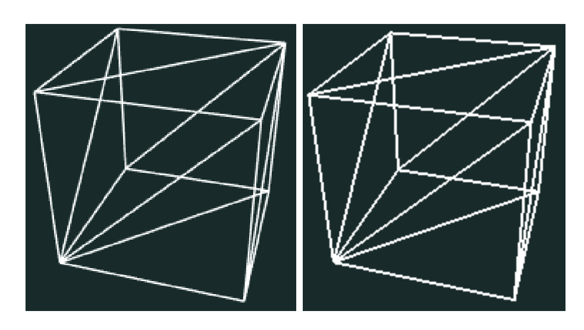


Figura 10.4: Comparación del resultado de aplicar MSAA (izquierda) y no aplicarlo (derecha).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

10.1.2. Corrección gamma

Los monitores no proporcionan una respuesta lineal respecto a los valores de intensidad de los píxeles. Esto produce que veamos las imágenes un poco más oscuras o no tan brillantes como realmente deberían observarse. En ciertas aplicaciones, es habitual que se ofrezca como opción al usuario poder realizar este ajuste de manera manual y así corregir el problema. En la figura 10.5, la curva CRT gamma muestra la respuesta del monitor para cada valor de intensidad de un píxel. La curva corrección gamma representa la intensidad del píxel necesaria para que el monitor presente la respuesta lineal, representada por la línea recta.

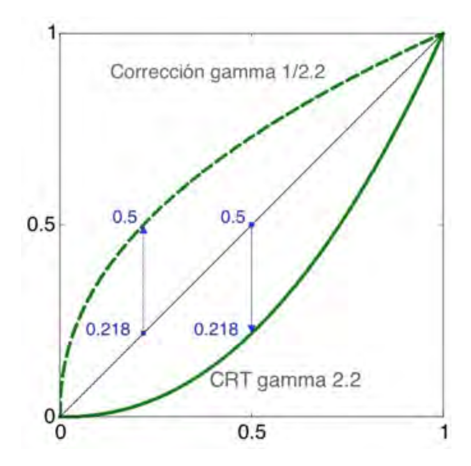


Figura 10.5: Esquema de funcionamiento de la correción gamma

Si la intensidad percibida es proporcional a la intensidad del píxel elevado a γ, $P = I^{\gamma}$, por lo que la correción gamma consiste en contrarrestar este efecto así:

$$P = (I^{\frac{1}{\gamma}})^{\gamma} \tag{10.2}$$

Esta operación se implementaría en el *shader* de fragmentos, tal y como figura en el listado 10.1 (véase ejemplo *c10/gamma.html*). La figura 10.6 muestra dos resultados obtenidos con valores de gamma 1.0 y 2.2.

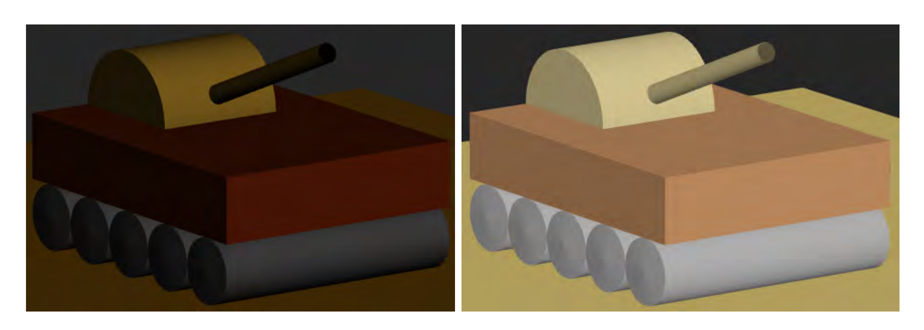


Figura 10.6: Ejemplos de corrección gamma: 1.0 (izquierda) y 2.2 (derecha)

ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 10.1: *Shader* de fragmentos para la correción gamma

```
...
uniform float Gamma;

void main() {
  ...
  vec3  myColor     = phong(n,L,V);
  float gammaFactor = 1.0 / Gamma;

  myColor.r         = pow( myColor.r, gammaFactor);
  myColor.g         = pow( myColor.g, gammaFactor);
  myColor.b         = pow( myColor.b, gammaFactor);

  fragmentColor     = vec4( myColor, 1.0);

}
```

10.2. Posproceso de imagen

En esta sección se consideran algunas técnicas de tratamiento de imágenes que se realizan a modo de posproceso del resultado de síntesis. Por ejemplo, las imágenes que se muestran en la figura 10.7 son el resultado del mismo proceso de síntesis y la diferencia es que, una vez obtenido el color del fragmento, se ha realizado alguna operación que se aplica por igual a todos los fragmentos de la imagen.



Figura 10.7: Ejemplos de posproceso de imagen

10.2.1. Brillo

La modificación del brillo de una imagen es un efecto muy sencillo. Consiste en escalar el color de cada fragmento por un valor $Brillo$. Si dicho valor es 1, la imagen no se altera, si es mayor que 1 se aumentará el brillo, y si es menor se disminuirá (véase figura 10.8). El listado 10.2 muestra el código que habría que incluir en el *shader* de fragmentos para modificar el brillo (véase ejemplo *c10/postproceso.html*).

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Listado 10.2: Modificación del brillo de una imagen

```
fragmentColor = vec4(miColor * Brillo, 1.0);
```



Figura 10.8: Ejemplos de modificación del brillo de la imagen con factores de escala 0.9, 1.2 y 1.5

10.2.2. Contraste

La alteración del contraste de la imagen se obtiene como resultado de mezclar dos colores, uno es el color obtenido como color del fragmento y el otro es el valor de luminancia media (véase figura 10.9). La variable *Contraste* se utiliza para dar más peso a un valor u otro (véase listado 10.3 y el ejemplo *c10/postproceso.html*).

Listado 10.3: Modificación del contraste de una imagen

```
vec3 LumiMedia = vec3(0.5, 0.5, 0.5);
fragmentColor = vec4(mix(LumiMedia, miColor, Contraste),1.0);
```

10.2.3. Saturación

La saturación es una mezcla del color del fragmento con un valor de intensidad específico de cada píxel (véase figura 10.10). Observa en el listado 10.4 las operaciones habituales para modificar la saturación que se pueden encontrar también en el ejemplo *c10/postproceso.html*.

Listado 10.4: Modificación de la saturación de la imagen

```
vec3 lumCoef      = vec3(0.2125, 0.7154, 0.0721);
vec3 Intensidad   = vec3(dot(miColor, lumCoef));
fragmentColor     = vec4(mix(Intensidad, miColor, Saturacion), 1.0);
```

10.2.4. Negativo

Otro ejemplo muy sencillo es la obtención del negativo de una imagen (véase figura 10.11). Únicamente hay que asignar como color final del fragmento el resultado de restarle a 1 su valor de color original. En el listado 10.5 se muestra el código correspondiente al cálculo del negativo del fragmento.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151



Figura 10.9: Ejemplos de modificación del contraste de la imagen: 0.5, 0.75 y 1.0



Figura 10.10: Ejemplos de modificación de la saturación de la imagen: 0.2, 0.5 y 0.8

Listado 10.5: Negativo del fragmento

```
fragmentColor = vec4(1.0 - miColor, 1.0);
```



Figura 10.11: Ejemplo del resultado del negativo de la imagen

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

10.2.5. Escala de grises

También es muy fácil la obtención de la imagen en escala de grises (véase figura 10.12). Lo más sencillo sería asignar como color final del fragmento el resultado de la media de sus tres componentes. Sin embargo, lo habitual es combinar las componentes de color con unos determinados pesos siguiendo la recomendación *ITU-R BT.709*. El listado 10.6 muestra el código correspondiente al cálculo del color del fragmento.

Listado 10.6: Cálculo de la imagen en escala de grises

```
float lum= 0.2125*miColor.r + 0.7154*miColor.g + 0.0721*miColor.b;
fragmentColor = vec4(vec3(lum), miColor.a);
```



Figura 10.12: Ejemplo del resultado de la imagen en escala de grises

10.2.6. Convolución

La convolución es una operación matemática fundamental en procesamiento de imágenes. Consiste en calcular para cada píxel la suma de productos entre la imagen fuente y una matriz mucho más pequeña a la que se denomina filtro de convolución. Lo que la operación de convolución realice depende de los valores del filtro. Para un filtro de dimensión $m \times n$ la operación es:

$$Res(x,y) = \sum_{j=0}^{n-1} \sum_{i=0}^{m-1} Img(x+(i-\frac{m-1}{2}), y+(j-\frac{n-1}{2})) \cdot Filtro(i,j) \quad (10.3)$$

Realizar esta operación con WebGL requiere que la imagen sintética se genere en primer lugar y se almacene después como textura para que en un segundo dibujado se pueda aplicar la operación de convolución sobre la imagen ya generada. Por otra parte, si la operación de la convolución sobrepasa los límites de la imagen, se utilizan los mismos parámetros que se indicaron para especificar el comportamiento de la aplicación de texturas fuera del rango $[0, 1]$.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

Las operaciones más habituales son el *blurring*, el *sharpening* y la detección de bordes, entre otras. La tabla 10.1 muestra ejemplos de filtros de suavizado o *blurring*, nitidez o *sharpening* y detección de bordes (véase figura 10.13). El ejemplo *c10/bordes.html* incluye una implementación de este último filtro.

1	1	1
1	1	1
1	1	1

(a)

0	-1	0
-1	5	-1
0	-1	0

(b)

-1	-1	-1
-1	8	-1
-1	-1	-1

(c)

Tabla 10.1: Filtros de convolución: (a) suavizado, (b) nitidez y (c) detección de bordes

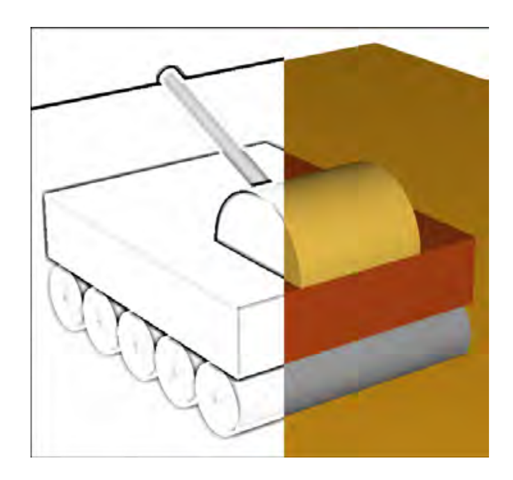


Figura 10.13: Ejemplo de resultado de la operación de convolución con el filtro de detección de bordes en la parte izquierda de la imagen

Ejercicios

▶ **10.1** Entre las operaciones típicas para el procesado de imágenes se encuentran las operaciones de volteo horizontal y vertical. ¿Cómo implementarías dichas operaciones?

▶ **10.2** Combina la operación de detección de bordes junto con el sombreado cómic visto en la sección 5.6 para obtener un resultado aún más similar al de un verdadero cómic.

10.3. Transformaciones

La transformación geométrica de una imagen se realiza en tres pasos:

1. Leer la imagen y crear un objeto textura con ella.

2. Definir un polígono sobre el que pegar la textura.

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

3. Escribir un *shader* de vértices que realice la operación geométrica deseada.

El paso 1 ha sido descrito en el capítulo 6. Para realizar el paso 2, un simple cuadrado de lado unidad es suficiente, no importa si la imagen no tiene esta proporción. En el paso 3, el *shader* de vértices debe transformar los vértices del rectángulo de acuerdo a la transformación geométrica requerida. Por ejemplo, para ampliar la imagen solo hay que multiplicar los vértices por un factor de escala superior a 1, y para reducirla el factor de escala debe estar entre 0 y 1. Es en este paso donde también se debe dar la proporción adecuada al polígono de acuerdo a la proporción de la imagen.

Otra transformación a realizar en el *shader* de vértices es el *warping*, es decir, la modificación de la imagen de manera que la distorsión sea perceptible. Esta técnica se realiza definiendo una malla de polígonos en lugar de un único rectángulo y modificando los vértices de manera conveniente (véase figura 10.14).



Figura 10.14: *Warping* de una imagen: imagen original en la izquierda, malla modificada en la imagen del centro y resultado en la imagen de la derecha

Como una extensión de la técnica anterior se podría realizar el *morphing* de dos imágenes. Dadas dos imágenes de entrada, hay que obtener como resultado una secuencia de imágenes que transforma una de las imágenes de entrada en la otra. Para esto se define una malla de polígonos sobre cada una de las imágenes y el *morphing* se consigue mediante la transformación de los vértices de una malla a las posiciones de los vértices de la otra malla, al mismo tiempo que el color definitivo de cada píxel se obtiene con una función de mezcla.

CUESTIONES

▶ **10.1** En el método de *antialiasing Full-Scene Anti-Aliasing* o FSAA, ¿cuántas veces se ejecutará el *shader* de fragmentos para cada píxel de la imagen final? ¿por qué?

▶ **10.2** Si al *shader* de fragmentos que estemos utilizando le añadimos la corrección gamma y fijamos un valor de gamma igual a 1.0, ¿cómo crees que afectará a la imagen final?

▶ **10.3** Utilizando GLSL se desea implementar un filtro de suavizado de imágenes mediante la operación de convolución y tamaño 5×5. ¿Qué ocurre cuando al aplicar la operación de convolución a los píxeles de los bordes de la imagen se intenta acceder a píxeles que se encuentren más allá de las propias dimensiones de la imagen?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4

Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151

▶ **10.4** Se quiere proporcionar al usuario la posibilidad de modificar el brillo de las imágenes generadas con WebGL. Se implementa un método en dos pasos que consiste en dibujar contra un FBO en el primer paso, y utilizar este resultado como textura de un único polígono que se dibuja en un segundo paso ya al *framebuffer* por defecto. ¿Qué atributos por vértice se han de proporcionar a la hora de dibujar este único polígono? Indica aquellos atributos que resulten estrictamente necesarios y justifica la respuesta.

▶ **10.5** Imagina que deseas modificar el brillo de la imagen. Durante el proceso de creación de la imagen defines en el *shader* de fragmentos una variable *uniform* que contendrá el factor de brillo. Si tu escena contiene objetos transparentes, ¿qué cuidado has de tener a la hora de operar con el factor de brillo?

▶ **10.6** La operación de convolución no se puede hacer al mismo tiempo que se obtiene la imagen sintética, se ha de realizar sobre la imagen sintética completamente calculada, ¿por qué?

José Ribelles y Ángeles López
ISBN: 978-84-17429-87-4
Informática Gráfica (2ª edición)
DOI: http://dx.doi.org/10.6035/Sapientia151